Escuela Superior de Ingeniería

Ingeniería Técnica en Informática de Gestión

Generación de software complementario para la integración de pruebas metamórficas en composiciones de servicios en WS-BPEL

María Azahara Camacho Magriñán

Cádiz, Septiembre de 2011

Escuela Superior de Ingeniería

Ingeniería Técnica en Informática de Gestión

Generación de software complementario para la integración de pruebas metamórficas en composiciones de servicios en WS-BPEL

- Departamento: Lenguajes y Sistemas Informáticos
- Directoras del proyecto: María del Carmen de Castro Cabrera, Inmaculada Medina Bulo
- Autora del proyecto: María Azahara Camacho Magriñán

Cádiz, Septiembre de 2011

Fdo: María Azahara Camacho Magriñán

Agradecimientos

Quisiera agradecer al grupo de investigación UCASE la oportunidad que me han dado al poder comenzar mi andadura en el mundo de la investigación junto a ellos. En especial, a Mª del Carmen de Castro Cabrera por todos los conocimientos y tiempo que me ha brindado, y a Inmaculada Medina Bulo por la ayuda prestada en los buenos y malos momentos.

A todos aquellos que alguna vez creyeron en mí y gracias a su confianza me dieron fuerzas para conseguir mis objetivos.

A los familiares y amigos que por alguna circunstancia nos dejaron. Gracias a Dios, me dieron fuerzas para seguir adelante día tras día para no caer en la tentación de abandonar.

Y por último, pero no por ello menos importantes, a mi familia y amigos. Los dos pilares fundamentales de mi vida, mis columnas de Hércules. Sólo ellos saben el trabajo y esfuerzo que he invertido en este trabajo. Por todo lo que me habéis dado y soportado, gracias.

No es más grande aquel que nunca falla,
sino el que nunca se da por vencido.

Índice general

Índice de figuras

1 Introducción

1.1. Pruebas metamórficas

Todo programa necesita pasar una serie de pruebas a la hora de comprobar su corrección y fiabilidad. Si dicho programa puede ser comprobado, se clasificará como *probable*, y en caso contrario, *no probable*.

Podemos considerar un programa probable aquel que dispone de algún mecanismo para comprobar su corrección. Y un programa no probable aquel que cumple alguna de estas dos condiciones:

- No existe un oráculo para dicho programa.
- Su comprobación teórica es posible, pero, en la práctica es muy difícil o prácticamente imposible comprobar la corrección de su salida.

El término *programa no probable* se utiliza sobre todo cuando no podemos determinar si la salida es o no correcta, o cuando tenemos que gastar una gran cantidad de recursos haciendo dicha comprobación. La figura del oráculo es fundamental, siendo un oráculo un mecanismo que comprueba la corrección de una salida respecto de unos datos de entrada, pudiendo ser desde una técnica, otro programa o un conjunto de personas expertas en el tema que se decantan por una solución u otra [21].

Por tanto, para poder comprobar la corrección de los programas no probables hay que buscar un método que lo haga. Éste es el propósito de las pruebas metamórficas, son un método que se ha propuesto para poner a prueba aquellos programas en los que no interviene un oráculo. Estudiamos una serie de casos de prueba, que según el resultado que den a la hora de aplicarles las pruebas, podrán ser exitosos o erróneos. De forma que con ellas descubriremos errores específicos que no se detectan con los casos de prueba exitosos y nos permitirá a su vez realizar pruebas computacionalmente costosas, que no podíamos realizar con los métodos tradicionales.

Los resultados obtenidos a partir de las pruebas metamórficas pueden ser utilizados para diseñar nuevos casos de prueba, y siguiendo una estrategia adecuada, podremos conseguir una mayor confianza en el software estudiado. Estos nuevos casos de prueba no se basan en complicadas teorías, sino, más bien en la experiencia en programación que a partir de fallos usuales en los programadores, nos puede permitir generar casos de prueba . Entre estos dos casos de prueba, el original y el que obtenemos, se establece una relación en la cual están relacionadas tanto la entrada como la salida de ambos casos. Estas relaciones que se establecen entre los casos de prueba son conocidas como *relaciones metamórficas* [6].

Las relaciones metamórficas (MR) son una serie de propiedades que se definen sobre un conjunto de datos de entrada y salida, y que se aplican sobre los casos de prueba a los que pertenecen dichos datos para comprobar su funcionamiento. Estas relaciones son las que nos ayudan a obtener nuevos casos de prueba, ya que sólo tenemos que generar un caso de prueba que cumpla las condiciones que se establecen en dicha relación [23]. Para ilustrar mejor esta funcionalidad de las relaciones metamórficas, vamos a considerar un programa que se llame *Suma* el cual se encargue de sumar una serie de enteros.

Por ejemplo, tenemos como entrada una lista de enteros:

$$enteros1 = \langle 1, 2, 5, 3, 2, 4, 3 \rangle$$

El resultado que obtenemos al aplicarle la función *Suma* a enteros1 es 20. Pues bien, si utlizáramos una segunda función que ordenara los elementos que contiene la lista, *Ordena*, obtendríamos un nuevo caso de prueba porque podríamos asegurar que la salida de ambos casos es idéntica al tratarse de los mismos sumandos. Es decir, que al aplicar *Ordena* a enteros1, obtendríamos como resultado:

$$enteros2 = \langle 1, 2, 2, 3, 3, 4, 5 \rangle$$

Y si aplicáramos la funcion *Suma* a enteros2, obtendríamos de nuevo como resultado el valor 20. De forma que podemos decir que *Suma(enteros1) = Suma(enteros2)* y de aquí, podríamos obtener una relación metamórfica de la siguiente estructura:

$$MR_1 \equiv (\exists lista2 = Ordena(lista1)) \rightarrow Suma(lista2) = Suma(lista1)$$

Donde lista1 es la lista de enteros original que tiene como entrada la función *Ordena* y lista2 es la lista de enteros que obtenemos como salida al aplicar la función *Ordena*. La función que está asociada a la relación metamórfica, y que por tanto, es la que establece la corrección de los casos de prueba es la función *Suma*, la cual nos permitirá decir si el programa que estamos estudiando tiene un error o no.

Pero, para detectar fallos en grandes programas, una relación metamórfica es insuficiente. Por ello, la técnica empleada en las pruebas metamórficas es disponer de un conjunto de casos de prueba previos y un conjunto de relaciones metamórficas. De forma que al ejecutar el programa, habrá casos de prueba en los que se detecte errores (*casos de prueba exitosos*) y otros en los que no (*casos de prueba erróneos*). Utilizando nosotros los casos de prueba exitosos para generar el nuevo conjunto de casos de prueba (*casos de prueba siguientes*) que nos permitirá seguir con el estudio del programa. Los pasos que hay que llevar a cabo para poder aplicar correctamente las pruebas metamórficas son los siguientes:

1. Generar un conjunto de casos de prueba previo con el que comenzar el estudio.
2. Escoger los casos de prueba exitosos, formando con ellos *el conjunto de casos de prueba inicial*.
3. Seleccionar las relaciones metamórficas adecuadas teniendo en cuenta el problema a resolver y el algoritmo que en él se emplea.
4. Generar el conjunto de casos de prueba siguientes aplicando las relaciones metamórficas seleccionadas en el paso previo.
5. Ejecutar el programa con los casos de prueba inicial y siguiente.

6. Comparar los resultados.

7. Mejorar el programa corrigiendo los posibles errores detectados, seleccionar nuevos casos de prueba y utilizar relaciones metamórficas mejoradas para la siguiente iteración.

La representación gráfica de estos pasos podemos verlos en el siguiente esquema:

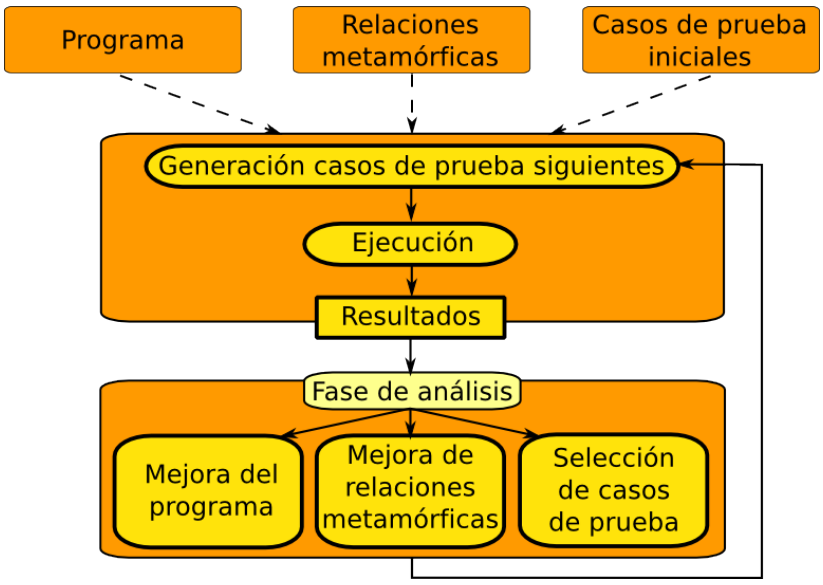

Figura 1.1: Diagrama de aplicación de MT

Dicho esquema se ha extraído del artículo "Una arquitectura basada en pruebas metamórficas para composiciones de servicios WS-BPEL" [5], en el cual se encuentra todo muy detallado.

Todos los pasos se pueden llevar a cabo en diversos lenguajes de programación. Nosotros concretamente vamos a centrarnos en el lenguaje de composición de servicios web WS-BPEL, Web Services-Business Process Execution Language, en castellano, Lenguaje de Ejecución de Procesos de Negocio con Servicios Web. Este lenguaje está basado en XML y está diseñado para el control centralizado de la invocación de diferentes servicios web, añadiéndole una lógica de negocio.

Una de las razones por la que estudiamos este lenguaje es porque permite la programación a gran escala. Esto es, que permite trabajar con grandes procesos de desarrollo, evolución y mantenimiento. Es un lenguaje de alto nivel que amplía el concepto de servicio, ya que proporciona métodos de definición y soporte para flujos de trabajo y procesos de negocio.

Pero BPEL depende del uso de WSDL (Web Services Description Language) para describir los mensajes entrantes y salientes entre los diferentes servicios. Es un formato en XML que permite describir la interfaz pública a los servicios web, describiendo los requisitos del protocolo y los formatos de los mensajes necesarios para que los servicios puedan interactuar entre

ellos. De forma que ambas tecnologías juntas, permiten trabajar correctamente a los diferentes servicios que participan en la composición.

Y para poder representar los diferentes casos de prueba que vamos a estudiar, vamos a utilizar BPTS (BPELUnit Test Suite). Esta tecnología nos permite reunir un conjunto de casos de prueba para BPELUnit, que es un framework de pruebas unitarias que realiza simulaciones automatizadas de la vida real, utilizando pruebas de caja blanca. Es decir, pruebas centradas en la funcionalidad del software, implicando una gran vinculación con el código fuente [16].

De forma que gracias a esta tecnología, podemos representar todos los casos de prueba que generemos a lo largo de nuestro estudio. En nuestro caso, vamos a utilizar BPTS basados en plantillas, que nos permite simplificar la complejidad del código y guardar en un fichero de tipo CSV, los diferentes casos de prueba que vamos a probar, de forma independiente.

1.2. Objetivos

Una vez explicados todos los pasos necesarios para aplicar las pruebas metamórficas, ya podemos pasar a explicar cuáles son los objetivos de este Proyecto Fin de Carrera. Se centra particularmente en los pasos dos y tres del proceso, es decir, en la selección de las relaciones metamórficas y en la generación del conjunto de casos de prueba siguientes utilizando las MR. Pero, aplicándolo al lenguaje de servicios web WS-BPEL.

La selección de las relaciones metamórficas de cada composición que vamos a estudiar es un proceso que se basa en todas las tecnologías implicadas, pero en particular, en el estudio de los archivos BPEL que son los que reflejan la lógica de la composición y en los archivos BPTS que son los que reflejan los casos de prueba que se van a aplicar a dicha composición.

A partir de los BPEL, obtendremos una serie de propiedades importantes donde podremos ver las diferentes variables y agentes que están implicados en la composición, los condicionales que hay, el comportamiento de los agentes ante ellos, etcétera. Con el estudio de los archivos BPTS, podremos concretar más cuáles son los datos con los que trabaja la composición y así, obtener los distintos casos de prueba que conformarán el conjunto de casos de prueba original.

Otro de los objetivos que queremos conseguir con este proyecto, es desarrollar los archivos *BPTS basados en plantillas* de cada una de las composiciones que vamos a estudiar. Esto es, simplificar todos los casos de prueba que disponemos en un BPTS original, en uno solo. Obteniendo así una plantilla en la que aplicar los diversos casos de prueba que guardaremos en un fichero CSV. Para conseguir estas plantillas, haremos uso de expresiones *X-Path* y de código *Velocity* que nos permitirán describir asignaciones, condiciones booleanas, etcétera y expresar los diferentes condicionales que se deben cumplir en determinadas partes del caso de prueba, respectivamente.

La generación de los nuevos casos de prueba a partir de las relaciones metamórficas es un

proceso cuya complejidad es directamente proporcional al número de condicionales que conforman la lógica de la relación. Normalmente, en composiciones con una cierta complejidad, las relaciones metamórficas que obtenemos de ellas suelen tener bastantes condiciones, por lo que generar varios casos de prueba puede ser una tarea que conlleve bastante tiempo y trabajo.

Por ello, la importancia del siguiente objetivo que es la automatización de la obtención de nuevos casos de prueba. Esta automatización consiste en reflejar toda la lógica de las diversas relaciones metamórficas que tenemos de una composición, y a partir del conjunto de casos de prueba originales y aplicándoles a cada uno de los casos los cambios lógicos y/o numéricos correspondientes, obtenemos el conjunto de casos de prueba siguientes.

Por último, ejecutaremos el programa junto a los casos de prueba originales y siguientes, y compararemos las salidas obtenidas. Siendo nuestro objetivo el detectar errores que anteriormente no habíamos encontrados. En la mayoría de las situaciones tendremos resultados exitosos, pero, en otros casos no obtendremos nada nuevo debido a diversas razones que ya explicaremos más adelante de forma más detallada.

1.3. Definiciones y acrónimos

Antes de comenzar con el desarrollo del proyecto, vamos a dar una serie de definiciones y acrónimos ordenados alfabéticamente que serán utilizados a lo largo del documento y que servirán para comprender mejor las diversas explicaciones y conceptos que queremos reflejar.

Muchos de ellos están relacionados entre sí, ya que algunos abarcan conceptos más genéricos y otros más particulares que necesitan una explicación añadida. También se va a tratar desde conceptos teóricos hasta conceptos relacionados con la práctica y las tecnologías utilizadas.

- BPELUnit: es un framework de pruebas unitarias que hace simulaciones de la vida real, automatizado y realiza pruebas de caja blanca con las composiciones BPEL [16].
- BPTS: BPELUnit Test Suite, conjunto de casos de prueba para BPELUnit recogidos en un fichero XML que define qué proceso se va a probar y cómo.
- BPTS basado en plantillas: documento BPTS en el que se encuentran simplificados todos los casos de prueba que disponemos en un BPTS original (en el cual está basado), en un solo caso. Obteniendo así una plantilla en la que aplicar los diversos casos de prueba que guardaremos en un fichero CSV.
- CSV: comma-separated values, es un tipo de documento de formato sencillo que nos permite representar datos en forma de tabla, donde las columnas se separan por comas y las filas por saltos de líneas. En nuestro caso, representaremos los datos de los casos de prueba en ficheros CSV.
- GAmera: sistema que genera y ejecuta automáticamente mutantes para composiciones de servicios web en WS-BPEL [9].

- Mockup: servicio simulado implementado con un comportamiento predefinido y que se suele emplear en pruebas que utilicen servicios costosos a los que no podemos acceder. No tienen una lógica establecida y simplemente se limitan a responder con mensajes predefinidos o fallando si es así como se le especifica.

- Mutante: programa que se genera a partir de un programa original, el cual contiene algún error. Se generan aplicando al código fuente una serie de reglas definidas a través de los operadores de mutación.

- Operadores de mutación: conjunto de reglas que añaden pequeños errores a los programas a los que se les aplica, basados estos errores en los que suelen cometer habitualmente los programadores, pero, mantiendo su validez sintáctica.

- Oráculo: mecanismo que comprueba la corrección de una salida respecto de unos datos de entrada, pudiendo ser desde una técnica, otro programa o un conjunto de personas expertas en el tema que se decantan por una solución u otra.

- Pruebas metamórficas: Metamorphic Testing (MT), método que se ha propuesto para poner a prueba aquellos programas en los que no interviene un oráculo y que está basado en el uso de relaciones metamórficas.

- Relaciones metamórficas: Metamorphic Relations (MR), son una serie de propiedades que se definen sobre un conjunto de datos de entrada y salida, y que se aplican sobre los casos de prueba a los que pertenecen dichos datos para comprobar su funcionamiento. Estas relaciones son las que nos ayudan a obtener nuevos casos de prueba.

- Servicios web: son sistemas software que utilizan un conjunto de protocolos y estándares para permitir la interacción máquina-a-máquina sobre una red. Su interfaz está descrita en WSDL y el resto de sistemas pueden comunicarse con él a través de mensajes SOAP. Se pueden utilizar estos servicios web para intercambiar datos en redes de ordenadores [10].

- SOAP: Simple Object Access Protocol, es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.

- Takuan: sistema que permite generar dinámicamente invariantes observados en una composición de servicios web con WS-BPEL [20].

- Velocity: es un motor de plantillas basado en Java. Le permite a los diseñadores de páginas hacer referencia a métodos definidos dentro del código Java. Se puede utilizar para crear páginas web, SQL, PostScript y cualquier otro tipo de salida de plantillas; como un componente integrado en otros sistemas...

- WS-BPEL: Web Services Business Process Execution Language, es un lenguaje diseñado para especificar servicios web. Está basado en XML y puede ser referenciado también como BPEL [19].

- WSDL: Web Services Description Language, es un lenguaje basado en XML que describe servicios web, es decir, los requisitos del protocolo y los formatos de los mensajes para interactuar con los servicios y agentes.
- XML: eXtensible Markup Language, lenguaje de etiquetado extensible muy simple, pero estricto, que nos permite estructurar, almacenar e intercambiar una gran cantidad de datos. Las tecnologías XML son un conjunto de módulos que ofrecen servicios a las peticiones más frecuentes de los usuarios.
- XPath: XML Path Language, lenguaje que nos permite construir expresiones que recorran y procesen documentos XML. Es una idea parecida a las expresiones regulares, pero, adaptado a documentos XML.

2 Desarrollo y calendario del proyecto

Una parte fundamental para el desarrollo de este trabajo ha sido la formación en las técnicas y herramientas que hemos necesitado. Para conseguir esta formación, los seminarios del grupo de investigación UCASE han jugado un papel muy importante. A continuación, vamos a describir todos los seminarios que han sido relevantes, las fases por las que ha pasado y la participación en un artículo para un congreso, fruto de este proyecto.

2.1. Seminarios

A lo largo del curso académico 2010-11, el grupo UCASE ha llevado a cabo una serie de seminarios en los cuales se han impartido unos conocimientos básicos y avanzados de determinadas herramientas, así como otros han sido destinados a la exposición del trabajo que estaba desarrollando cada uno. De forma cronológica, estos son los seminarios que han sido fundamentales para este trabajo:

- Viernes 1 de Octubre de 2010: Se acordó la composición WS-BPEL con la que íbamos a comenzar el estudio (LoanApproval) y las posibles composiciones que la seguirían.
- Viernes 22 de Octubre de 2010: Se presentan y resuelven las dudas relacionadas con las tecnologías implicadas en el estudio que habían surgido al comenzar con el trabajo.
- Viernes 29 de Octubre de 2010: Introducción a las pruebas de mutaciones en WS-BPEL y a la herramienta GAmera. Se explica en qué consisten las pruebas de mutaciones, los distintos mutantes que podemos encontrarnos a la hora de realizar las pruebas y la herramienta que nos permite trabajar con estos mutantes, MuBPEL. También se comenta el desarrollo del trabajo de GAmera, pero, no influye a nuestro trabajo sino al de otros compañeros. Para poder acceder a los distintos documentos y ficheros de los que dispone el grupo de investigación se me da de alta en la forja *https://neptuno.uca.es*.
- Viernes 12 de Noviembre de 2010: Introducción a las pruebas metamórficas. Se explica en qué consisten, los conceptos fundamentales y la relación que tienen con los mutantes. También se comenta la aplicación de estas pruebas metamórficas a las aplicaciones de servicios WS-BPEL.
- Martes 1 de Febrero de 2011: Se explican algunos casos de prueba de las composiciones WS-BPEL que se estudian en el grupo y parte de la tecnología que utilizan, BPTS.
- Martes 22 de Febrero de 2011: Comparativa entre los operadores de mutación de WS-BPEL y los de otros lenguajes. En este seminario se nos explica las pricipales diferencias que existen en los operadores de mutación entre distintos lenguajes. Nos ayuda a ver un poco mejor el funcionamiento de estos operadores en las pruebas de mutaciones.

- Martes 15 de Marzo de 2011: Expongo los primeros avances conseguidos hasta ahora, que era la obtención de tres casos de prueba nuevos que conseguían matar a dos de los mutantes de LoanApproval que anteriormente no se les había detectado ningún error. Se propone una automatización de la generación de casos de prueba que aplique las relaciones metamórficas correspondientes.

- Martes 29 de Marzo de 2011: Segunda exposición de mis avances, en este caso, se explican algunas de las relaciones metamórficas que he implementado en la automatización. Y muestro el fichero CSV que genera la primera versión de la aplicación.

- Lunes 4 de Abril de 2011: Taller sobre Lynx y Zotero, herramientas que nos servirán para la escritura de artículos y también, para la memoria del Proyecto Fin de Carrera.

- Martes 12 de Abril de 2011: Expongo los avances de mi parte del artículo de investigación para las Jornadas de Ciencia e Ingeniería de Servicios (JCIS). En él explicó un breve ejemplo de aplicación de las pruebas metamórficas y las ventajas que nos ofrece la automatización de la generación de casos de prueba.

- Martes 26 de Abril de 2011: Se plantea comenzar con el estudio de otra composición, MarketPlace. Como en este caso no se dispone de fichero BPTS basado en plantillas, hay que empezar con su implementación.

- Lunes 30 de Mayo de 2011: Tercera exposición de los avances en mi trabajo. Aquí ya funciona correctamente la aplicación de LoanApproval y explico el primer BPTS basado en plantillas que he implementado para la composición MarketPlace. Surge el problema del atributo “delaySequence” que no podemos adaptarlo a la plantilla.

- Miércoles 15 de Junio de 2011: Taller de Inkscape, herramienta gráfica que nos permitirá realizar gráficos con una gran calidad, al tratarse de imágenes vectoriales. Permite la transformación de ellos en archivos .zip, .png, .pdf, .tex, etcétera. Nos ayudará mucho para los gráficos que necesitemos en la memoria del Proyecto Fin de Carrera.

- Jueves 7 de Julio de 2011: Ya he implementado la primera versión de la aplicación de la composición MarketPlace, por tanto, se plantea comenzar con el estudio de la tercera composición, MetaSearch.

- Jueves 21 de Julio de 2011: Cuarta y última exposición de mis avances. Expongo las dudas que tengo en el funcionamiento de la composición MarketPlace debido al atributo “assume” del BPTS. Y también, las dudas respecto a la lógica de la composición MetaSearch, en la cual aparece de nuevo el inconveniente del atributo “delaySequence” y la adaptación a las plantillas de las condiciones y resultados de cada caso de prueba. Se propone una solución y queda pendiente como trabajo futuro.

Como explico en el último seminario, muchas cosas quedaron pendientes (sobre todo de la última composición). Por ello, en los capítulos posteriores haré referencia a estas dificultades que fueron surgiendo y cómo se solucionaron a lo largo del desarrollo del proyecto.

Pero uno de ellos en particular, el problema del atributo “delaySequence” ha sido uno de los más difíciles de resolver, ya que BPELUnit no nos permitía esa adaptación. Gracias al trabajo de Antonio García Domínguez se ha resuelto y finalmente el 3 de Agosto de 2011 ya se podía adaptar el atributo “delaySequence” a los ficheros BPTS basados en plantillas como “delay”.

2.2. Artículo JCIS 2011

JCIS es el resultado de la integración de las Jornadas Científico-Técnicas en Servicios Web y SOA (JSWEB) y el Taller sobre Procesos de Negocio e Ingeniería de Servicios (PNIS). En esta edición JCIS se organiza conjuntamente con JISBD y PROLE, bajo el auspicio de SISTEDES. Por tanto, encaja perfectamente con el tema que tratamos en este trabajo. Es por ello, que en el seno del grupo de investigación se decidió escribir un artículo en el que se presentara la propuesta de aplicación de pruebas metamórficas a composiciones de servicios en WS-BPEL con un esquema de implementación, una arquitectura y los avances del trabajo realizado hasta la fecha. Este trabajo fue aceptado en Junio de 2011 y ha sido presentado durante las JCIS 2011 (del 5 al 7 de Septiembre), en A Coruña. A continuación incluimos información relevante del mismo, así como el resumen:

Título: Una arquitectura basada en pruebas metamórficas para composiciones de servicios WS-BPEL.

Autores: Mª del Carmen de Castro Cabrera, Mª Azahara Camacho Magriñán, Inmaculada Medina Bulo y Manuel Palomo Duarte.

Resumen: Se propone el uso de las prueba metamórficas para el estudio y pruebas de las composiciones en WS-BPEL. Se explica la arquitectura propuesta para su implementación y un caso de estudio. A partir de los resultados, se muestran las ventajas del uso de dichas pruebas para la generación automática de nuevos casos de prueba que nos permitan detectar errores en la composiciones que anteriormente no se habían encontrado [5].

2.3. Fases de desarrollo del proyecto

Este trabajo, al tratarse de una implementación para temas de investigación, ha requerido muchas horas de estudio, de búsqueda y de pruebas. Vamos a dividir en bloques cada uno de los trabajos que se han ido llevando a cabo para conseguir como resultado este Proyecto Fin de Carrera. Pero, estos bloques no son fijos, es decir, que pueden solaparse en el tiempo. Al igual, que no es un requisito fundamental que se haya terminado con una fase para comenzar con otra, sino, que puede comenzarse en el momento que sea necesario.

2.3.1. Fase I: Toma de contacto

Inicio de Fase I: Octubre de 2010.

Fin de Fase I: Septiembre de 2011.

En esta fase veo por primera vez cómo funciona el mundo de la investigación: qué temas están ahora mismo en estudio en el grupo, cómo se reparten el trabajo, cómo se organiza la documentación de todos los temas... Pero, sobre todo, conozco de qué trata mi trabajo en particular. Me dedico sobre todo, a la lectura de artículos de investigación que tratan de temas como los servicios web [22, 4, 12], los mutantes [9], las pruebas metamórficas [13, 18, 3, 7, 17] y el lenguaje WS-BPEL y BPTS [2, 16, 11, 19].

Este estudio ha sido continuo durante todo el trabajo, pero, los primeros meses de comienzo han sido los más implicados en esta actividad.

2.3.2. Fase II: LoanApproval, estudio de la composición

Inicio de Fase II: Enero de 2011.

Fin de Fase II: Febrero de 2011.

Comienzo con el estudio de la primera composición que vamos a tratar con relaciones y pruebas metamórficas. Al tratarse de la primera, es la que pudo costarme algo más de trabajo a la hora de comprender la organización y funcionamiento del código WS-BPEL, BPTS y WSDL todo junto. Hago pruebas con MuBPEL y obtengo los mutantes que aún están vivos después de probarlos con el conjunto de casos de prueba original.

2.3.3. Fase III: LoanApproval, relaciones metamórficas

Inicio de Fase III: Febrero de 2011.

Fin de Fase III: Marzo de 2011.

Una vez que ya tengo claro cómo funciona la composición de LoanApproval y he obtenido los mutantes que están vivos, me dedico a desarrollar posibles relaciones metamórficas con las cuales podamos obtener casos de prueba que maten a esos mutantes. Para ello, estudiamos las diferencias existentes entre nuestro fichero BPEL original y el mutante al que queramos detectar el error, y desarrollamos una relación metamórfica que se base en esa diferencia. Luego, obtenemos algún caso de prueba "a mano" a partir de esta realación metamórfica y vemos que, en efecto, detecta nuevos errores.

2.3.4. Fase IV: LoanApproval, automatización

Inicio de Fase IV: Marzo de 2011.

Fin de Fase IV: Abril de 2011.

En esta fase me encargo de automatizar todas las relaciones metamórficas que hemos desarrollado en la fase anterior. Como no se me impone un lenguaje en particular para realizar la aplicación, la implemento en código C++.

2.3.5. Fase V: LoanApproval, pruebas con los nuevos resultados

Inicio de Fase V: Abril de 2011.

Fin de Fase V: Abril de 2011.

Una vez que ya consigo que la aplicación funcione correctamente y elimine los casos de prueba repetidos, me dispongo a probar el fichero que genera (data2.csv) con la composición en MuBPEL. Obtengo unos buenos resultados, ya que acabamos detectando nuevos errores que anteriormente no se detectaban.

2.3.6. Fase VI: MarketPlace, estudio de la composición

Inicio de Fase VI: Abril de 2011.

Fin de Fase VI: Mayo de 2011.

Una vez que ya he terminado con la composición LoanApproval, comienzo con el estudio de la segunda composición. Es más sencilla en cuanto a la lógica que tiene y a la extensión de código, pero, aparece la concurrencia y demora en el tiempo de respuesta. Cosas que no habían aparecido en la composición anterior. Probamos la composición con MuBPEL y comprobamos que sólo uno de los mutantes sigue vivo al aplicarle el conjunto de casos de prueba original.

2.3.7. Fase VII: MarketPlace, BPTS basado en plantillas

Inicio de Fase VII: Abril de 2011.

Fin de Fase VII: Agosto de 2011.

Para facilitar la aplicación de los casos de prueba, utilizamos los ficheros BPTS basados en plantillas. Para realizarla, nos basamos en la estructura que siguen los casos de prueba del BPTS original y nos ayudamos del BPTS basado en plantillas de la composición LoanApproval para comprobar que es correcta. Realmente, en Mayo ya estaba preparada para trabajar con casos de prueba que no tuvieran en cuenta la demora del tiempo, pero, debido a un problema con el atributo “delaySequence” no estuvo completada hasta Agosto.

2.3.8. Fase VIII: MarketPlace, relaciones metamórficas

Inicio de Fase VIII: Mayo de 2011.

Fin de Fase VIII: Mayo de 2011.

Como el inconveniente del "delaySequence" no era algo que interviniera de forma importante a la hora de desarrollar las relaciones metamórficas, comenzamos a estudiar las diferencias existentes entre el fichero BPEL original y el mutante que sigue vivo. Comprobamos que la única diferencia que existe entre ellos es un cambio de orden a la hora de trabajar una respuesta, por tanto, desarrollamos posibles relaciones que pudieran detectar algún error en ese mutante.

2.3.9. Fase IX: MarketPlace, automatización

Inicio de Fase IX: Mayo de 2011.

Fin de Fase IX: Agosto de 2011.

Como no obtenemos ningún resultado relevante a la hora de crear casos de prueba "a mano", probamos a automatizarlo para así aumentar el número de nuevos casos de prueba e intentar detectar el error. Al igual que en la composición anterior, se implementó en C++, pero, como la aplicación está tan ligada al fichero BPTS basado en plantillas y éste no estuvo terminado hasta Agosto, la aplicación tampoco estuvo completamente terminada hasta ese mismo mes. En Mayo, implementé una primera versión que trabajaba con casos de prueba que no trataban la demora de respuesta de cada agente.

2.3.10. Fase X: MarketPlace, pruebas con los nuevos resultados

Inicio de Fase X: Mayo de 2011.

Fin de Fase X: Agosto de 2011.

Esta fase tiene dos partes distintas. Una en la que realizamos los estudios con los casos obtenidos en la primera versión de la aplicación y otra, en la que se realizan las pruebas con los casos de la versión correcta de la aplicación. Probamos todos los casos que obtenemos de la automatización y comprobamos los resultados de aplicarlos al mutante que seguía vivo.

2.3.11. Fase XI: MetaSearch, estudio de la composición

Inicio de Fase XI: Julio de 2011.

Fin de Fase XI: Agosto de 2011.

Como en Julio el inconveniente del "delaySequence" no tenían aún ninguna solución, comencé con el estudio de MetaSearch. Me encontré con una composición que mezclaba todo lo visto anteriormente, incluyendo la demora de tiempo en la respuesta de los agentes implicados. Utilicé MuBPEL junto al fichero BPTS en el que se recogían los casos de pruebas originales y obtuvimos varios mutantes vivos como salida.

2.3.12. Fase XII: MetaSearch, BPTS basado en plantillas

Inicio de Fase XII: Julio de 2011.

Fin de Fase XII: Agosto de 2011.

Al igual que MarketPlace, esta composición tampoco dispone de un BPTS basado en plantillas, por lo que antes de buscar relaciones metamórficas, tuve que implementar un BPTS que permitiera aplicar los casos de prueba desde un fichero CSV. Tuvimos varios problemas, ya que al ser una composición tan compleja, tenía una gran variedad de posibles casos de prueba. De forma que estuve haciendo diversas pruebas con algunas versiones, hasta que finalmente llegué a una versión final que recogía la posibilidad de probar unos casos de pruebas más sencillos.

2.3.13. Fase XIII: MetaSearch, relaciones metamórficas

Inicio de Fase XIII: Agosto de 2011.

Fin de Fase XIII: Agosto de 2011.

Para desarrollar las relaciones metamórficas de esta composición, tuve que dedicarle más tiempo que el dedicado a las otras dos composiciones. El motivo, es que en ésta se podían aplicar cambios aritméticos o lógicos, pero a su vez, se podían aplicar cambios que afectaran a los resultados de cada caso de prueba, cambios en algunos de los datos proporcionados por el agente cliente, demoras de tiempo . . . Finalmente, obtuve algunas que pude aplicarlas al BPTS basado en plantillas, pero, otras no podía aplicarlas. Por lo que obtuve algunos casos a partir de ellas y los recogí en un BPTS nuevo llamado "MetaSearchTest-extendedAzahara.bpts".

2.3.14. Fase XIV: MetaSearch, automatización

Inicio de Fase XIV: Agosto de 2011.

Fin de Fase XIV: Agosto de 2011.

En esta fase implementé las relaciones metamórficas que desarrollé en la fase anterior que era posible aplicarlas al BPTS basado en plantillas, pero, al contrario que en las otras composiciones, lo implementé en lenguaje Java para así poder comprobar luego la validez del software en la plataforma "Sonar". Esta implementación llevó un poco más de trabajo que las dos anteriores, debido a que tenía una gran cantidad de condiciones para poder obtener un caso de prueba correcto.

2.3.15. Fase XV: MetaSearch, pruebas con los nuevos resultados

Inicio de Fase XIV: Agosto de 2011.

Fin de Fase XIV: Septiembre de 2011.

Una vez que ya acabé con la automatización de las relaciones, me puse a probar los casos de prueba que se habían generado, al igual que hice con las composiciones anteriores. En este caso, obtuvimos resultados relevantes, tanto por parte de los casos de prueba aplicados en la plantilla, como con los casos de prueba del nuevo BPTS.

2.3.16. Fase XVI: Sonar

Inicio de Fase XIV: Julio de 2011.

Fin de Fase XIV: Agosto de 2011.

Esta fase ocupa dos meses porque incluye la adaptación de las aplicaciones de LoanApproval y de MarketPlace al lenguaje Java. Una vez que ya las implementé y documenté correctamente, me encargué de añadirlas a la forja del grupo de investigación para que así pudiera estar en la plataforma "Sonar". Cuando ya se ejecutaba desde ella, me indicaba los fallos que contenía y cómo podía arreglarlos. De esta forma, he conseguido que la calidad del software mejore notablemente. Con la aplicación de MetaSearch no tuve problemas porque al estar implementada en Java, sólo tuve que añadirla a la forja y ver los resultados que indicaba la plataforma.

2.3.17. Fase XVII: Documentación

Inicio de Fase XIV: Julio de 2011.

Fin de Fase XIV: Septiembre de 2011.

Este bloque incluye todo el desarrollo de la memoria de este Proyecto Fin de Carrera. Incluyendo las correcciones propuestas por las tutoras, las mejoras del texto y la preparación de la presentación de dicho proyecto ante el Tribunal.

2.4. Diagrama de Gantt

En este apartado vamos a mostrar gráficamente, gracias a un diagrama de Gantt, el espacio temporal que han ocupado cada una de las fases que hemos explicado anteriormente. Podemos ver que al principio, las tareas las realicé de forma más secuencial, pero, ya una vez que acabé con el trabajo de la primera composición, el resto fue más fácil de llevar y ha habido tareas que las he podido realizar de forma paralela.

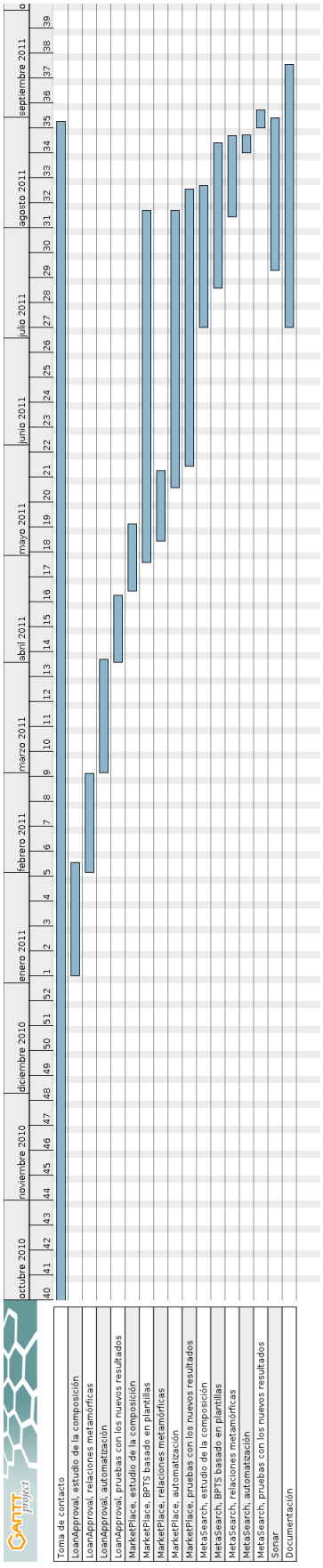

Figura 2.1: Diagrama de Gantt

2.5. Presupuesto del proyecto

Para poder realizar este proyecto se ha necesitado emplear una serie de costes y en función del trabajo que hayamos estado realizando, el coste ha sido distinto. A continuación vamos a explicar el coste que se ha necesitado para poder realizar cada una de las tareas, teniendo en cuenta que es un coste estimado, ya que las pruebas de los resultados también dependerán de las características del hardware que estemos utilizando. Para realizar todas las pruebas, he utilizado un ordenador que tiene un procesador Inter(R) Core(TM)2 Duo de 2.00GHz, una memoria de 2,0 GiB y en cuanto al sistema operativo, la versión 11.04 de Ubuntu.

Todas las composiciones han tenido un patrón común de estudio: una primera parte dedicada al análisis de la composición, otra enfocada al diseño de las relaciones metamórficas, a la implementación del software y a las pruebas de los nuevos casos de prueba con MuBPEL. Son cuatro partes diferentes que según la composición que hayamos estudiado habrá necesitado más o menos días. Vamos a estudiar cada composición y veremos las diferencias más relevantes entre las fases de las composiciones.

Vamos a asignarle a cada fase un número de días, teniendo en cuenta que cada día se le ha podido dedicar unas 6 horas de estudio de media, ya que al principio del proyecto compaginaba las asignaturas del tercer curso de la carrera con este estudio. Para verlo más claramente, cada composición va a tener un diagrama de barras en el que se diferencie cada fase.

2.5.1. Presupuesto de LoanApproval

En esta composición, las fases de análisis y de diseño de las relaciones metamórficas han sido las más extensas. La fase de análisis duró siete días, ya que, aunque no fuese una composición muy complicada, en ella nos encontramos por primera vez con el código BPEL y BPTS que fue lo que nos retrasó un poco más. También se estudió las diferencias entre los mutantes generados por MuBPEL y el fichero BPEL original, lo que dio pie a la fase de diseño de relaciones metamórficas. En ella se diseñaron las diferentes relaciones, según las diferencias encontradas en la fase anterior. Duró 10 días, en los cuales estudiamos cómo desarrollar una relación metamórfica a través del estudio de artículos y otras composiciones, y posteriomente, desarrollamos las relaciones metamórficas de LoanApproval.

La fase de implementación duró 6 días y abarcó el desarrollo de la aplicación que generaba los nuevos casos de prueba. Adaptamos las relaciones metamórficas a código C++ en un principio, pero más adelante, se implementó en código Java. Y por último, la fase de pruebas de estos nuevos resultados obtenidos con MuBPEL. Duró sólo 3 días, y en ellos estuvimos comparando los resultados obtenidos y viendo qué nuevos mutantes eran detectados. En la Figura 2.2 podemos ver cómo ser reparten estos días.

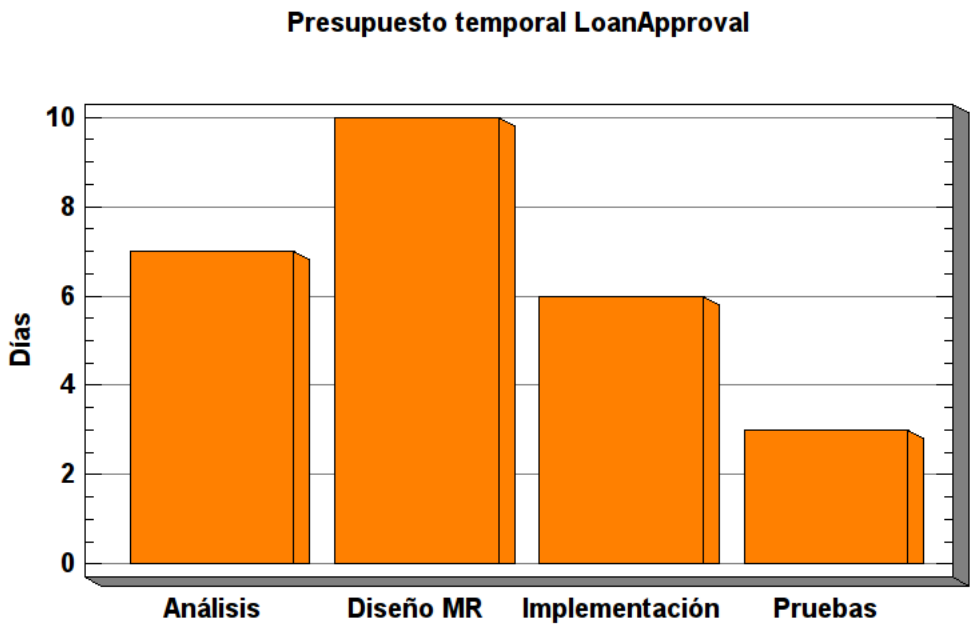

Figura 2.2: Presupuesto temporal de LoanApproval

2.5.2. Presupuesto de MarketPlace

El tiempo empleado en esta composición, tiene una proporción muy parecida al de la composición anterior. Las fases de análisis y diseño de las relaciones metamórficas son las que más tiempo han empleado. La fase de análisis duró cinco días, en este caso, se extendión un poco más debido al estudio de la concurrencia en BPEL y el nuevo atributo "delaySequence" en el fichero BPTS. Esto hizo que tuviera que hacer pruebas para ver cómo funcionaba realmente y cómo afectaba a los datos de cada caso de prueba que ya disponíamos.

En la fase de diseño de relaciones metamórficas se emplearon ocho días. Realmente, en un principio fueron menos, lo que ocurre es que aún seguían existiendo una serie de dudas con "delaySequence" que nos hizo comprobar de nuevo ciertos casos de prueba, pero, finalmente se desarrollaron las relaciones metamórficas correspondientes. La fase de implementación en este caso duró tres días, ya que la aplicación era mucho más sencilla de implementar. Tuvimos también que implementar el nuevo BPTS basado en plantillas, pero, no fue muy complicado de realizar teniendo en cuenta que disponíamos del BPTS basado en plantillas de la composición LoanApproval como guía. Al igual que anteriormente, implementamos primero la aplicación en C++ y más adelante se adaptó al lenguaje Java para que trabajara con la plataforma Sonar.

Por último, la fase de pruebas con MuBPEL llevó algo más de tiempo que con la composición LoanApproval. Como sólo nos teníamos que centrar en ese mutante que he comentado anteriormente, esperábamos obtener resultados relevantes al aplicar los casos de prueba que habíamos obtenido con la aplicación. Pero, no fue así, por lo que estuvimos estudiando y barajando la posibilidad de que se tratase de un mutante equivalente. En capítulos posteriores ex-

plicaremos estos estudios más detalladamente. En la Figura 2.3 podemos ver cómo ser reparten.

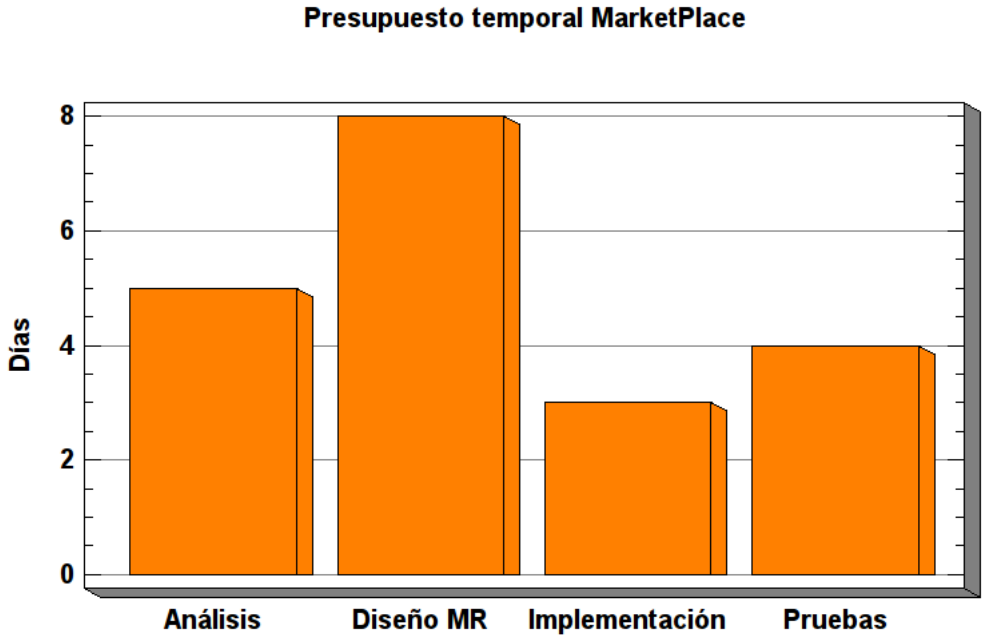

Figura 2.3: Presupuesto temporal de MarketPlace

2.5.3. Presupuesto de MetaSearch

Al igual que en las composiciones anteriores, la fase de análisis ha sido una de las más extensas, concretamente, ha durado quince días. En este caso, ha llevado tanto tiempo debido a la complejidad de la composición y a la gran cantidad de mutantes que genera MuBPEL. Muchos de estos mutantes fueron detectados por los casos de prueba que ya habían desarrollado los miembros del grupo de investigación, pero, otros muchos aun seguían vivos. Por lo que el estudio de las diferencias entre los mutantes y el fichero BPEL original fue mucho más lento que en las otras dos ocasiones. Al llevarme tanto tiempo con este estudio, algunas de las relaciones metamórficas fueron surgiendo casi a la vez, por lo que la fase de diseño de las relaciones metamórficas duró ocho días, obteniendo una gran cantidad de relaciones.

La fase de implementación duró seis días, y al igual que en la composición de MarketPlace, tuvimos que implementar el BPTS basado en plantillas y la aplicación que generaba el conjunto de los nuevos casos de prueba. En este caso fue implementada directamente en Java y como novedad respecto a las composiciones anteriores, se implementó un nuevo BPTS con casos de prueba que no eran adaptables a la plantilla.

Por último, la fase de pruebas con MuBPEL ha llevado bastante más tiempo que en el resto. Esto se ha debido por varios factores, en primer lugar, a la gran cantidad de nuevos casos de prueba generados, y luego, por el estudio de algunos mutantes que no eran detectados como

erróneos y que podían tratarse de nuevo de mutantes equivalentes. En capítulos posteriores se detallará este estudio. En la Figura 2.4 podemos ver cómo ser reparten los días en esta composición.

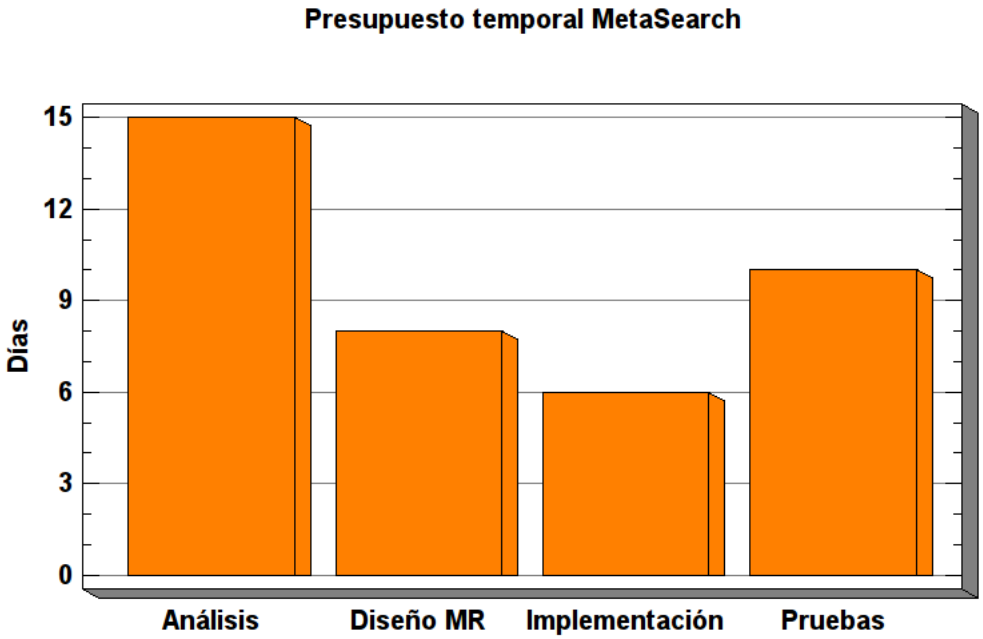

Figura 2.4: Presupuesto temporal de MetaSearch

3 Fundamentos

En este apartado, vamos a centrarnos en explicar cuáles han sido los estudios necesarios para poder comenzar el desarrollo del Proyecto Fin de Carrera. Tratando desde los conceptos que ya existían sobre las Pruebas Metamórficas y las Relaciones Metamórficas, hasta los diversos lenguajes que se van a utilizar para poder conseguir todos los objetivos propuestos en la Introducción.

3.1. Antecedentes

Como ya se ha comentado en la introducción de la memoria, las Pruebas Metamórficas (Metamorphic Testing, MT) son un método que se ha propuesto para poner a prueba aquellos programas en los que no interviene un oráculo y que está basado en el uso de relaciones metamórficas. El origen de esta definición, o más bien, del concepto de prueba metamórfica, se remonta a 1982 donde E.J. Weyuker [21] se plantea pruebas de software alternativas a las existentes que permitan resolver los problemas que en ese momento no se podían solventar por los métodos tradicionales. En este momento, se tantea el concepto de prueba metamórfica, pero, aún no se le conoce o no se le llama de esa manera.

Tiene que ser 16 años después, en 1998, cuando se reconoce a este concepto como Prueba Metamórfica [6]. Lo lleva a cabo T.Y.Chen y relaciona el concepto de *Metamorphic Testing* como una nueva técnica para generar casos de prueba que permiten comprobar programas que con las técnicas tradicionales no obtenían resultados relevantes. En años posteriores, Chen y otros autores como A.Gotlieb, Z.Q.Zhou o Z.Yang, han mostrado las aplicaciones de estas pruebas y la efectividad de las técnicas empleadas.

Una de las primeras aplicaciones de estas pruebas aparecen en 2002 de la mano del creador del nombre de este tipo de pruebas, T.Y.Chen [7]. En él se describe la aplicación para pruebas sobre programas numéricos en general, y concretamente, en programas que resuelven ecuaciones en derivadas parciales. Sus conclusiones en este artículo es que las pruebas metamórficas ayudan a resolver el problema del oráculo en pruebas para software numérico (algunos de estos programas no tienen un oráculo que compruebe su validez). Y otra conclusión más, es que las MT aplicadas a un caso en particular, pueden utilizarse en otros métodos numéricos.

Un año más tarde (2003), A.Gotlieb y A.Botella proponen automatizar las Pruebas Metamórficas mediante un prototipo llamado *AMT (Automated Metamorphic Testing)* que comprueba de forma automática las relaciones metamórficas [13]. Éste está implementado sobre un generador automático de casos de prueba llamado INKA, mostrando resultados experimentales. Las

conclusiones que obtienen es que esta técnica es muy útil e interesante para los programas declarados como *no probables*, y que además, las relaciones metamórficas elegidas influyen mucho en la velocidad de detección de errores.

En 2004, en un artículo de Z.Q.Zhou [23] se propone aplicar las MT en:

- Software que solucione problemas numéricos.
- Software que solucione problemas no numéricos como grafos, compiladores, software interactivo ...

Y hace una serie de aclaraciones para utilizar estas pruebas: la primera es que debe de combinarse las MT con otras estrategias de selección de casos de prueba, y la segunda aclaración es que es necesario y fundamental un conocimiento del dominio del problema para poder obtener una aplicación efectiva de las MT en dicho problema. Esto último ya se vio reflejado en una de las conclusiones de Gotlieb y Botella, ya que vieron que la velocidad de detección de errores se veía muy influenciada por las relaciones metamórficas elegidas. De forma que, tan importante es desarrollar buenas relaciones metamórficas, como conocer la influencia que pueden llegar a tener dichas relaciones en el conjunto del problema.

En ese mismo año, T.Y.Chen junto a otros autores, describe en un artículo la importancia de la elección de relaciones metamórficas efectivas mediante un experimento basado en un problema de cálculo de caminos de grafos [8]. Obtuvo diversas conclusiones, entre ellas que el uso de relaciones metamórficas basadas en parámetros críticos del algoritmo a utilizar, muestran una mayor capacidad para detectar los errores. De nuevo aparece la idea de que las pruebas metamórficas sirven para resolver los casos de prueba que tienen el problema del oráculo; y además, que estas pruebas son automatizables. Como podemos ver, en todos los estudios que se han llevado a cabo hasta ahora, el dominio del problema es un punto fundamental en el desarrollo de las pruebas metamórficas, ya que influyen tanto en la relevancia de los resultados obtenidos, como en el tiempo invertido en ello.

Luego, ha habido más artículos que han mostrado que el uso de las técnicas basadas en las pruebas metamórficas han tenido un gran éxito y que los resultados que se han obtenido han sido de gran ayuda para mostrar los avances conseguidos. Pero, también para corroborar las conclusiones y teorías desarrolladas en los artículos que se han explicado.

3.2. Lenguaje WS-BPEL

El lenguaje WS-BPEL (*Web Services Business Process Execution Language*) es un lenguaje diseñado para especificar servicios web. Está basado en XML y puede ser referenciado también como BPEL [19]. Ésta es una de las muchas definiciones que puede usarse para hablar de este lenguaje que está enfocado al mundo de los servicios web.

Todo proceso desarrollado en WS-BPEL tiene una serie de partes bien diferenciadas. En este apartado vamos a explicar cada una de ellas, basándonos en el fichero BPEL que forma parte de la composición LoanApproval (LoanApproval.bpel):

1. Una primera parte en la cual importamos de otros ficheros las definiciones de los mensajes entrantes y salientes de los servicios. Estará formado por tantos elementos "import" como ficheros wsdl tenga que utilizar. Su estructura es la siguiente:

```
<import importType="http://schemas.xmlsoap.org/wsdl/"
 location="ApprovalService.wsdl"
 namespace="http://j2ee.netbeans.org/wsdl/ApprovalService"/>

<import importType="http://schemas.xmlsoap.org/wsdl/"
 location="AssessorService.wsdl"
 namespace="http://j2ee.netbeans.org/wsdl/AssessorService"/>

<import importType="http://schemas.xmlsoap.org/wsdl/"
 location="LoanService.wsdl"
 namespace="http://j2ee.netbeans.org/wsdl/LoanService"/>
```

2. La segunda parte sirve para definir las relaciones con los socios externos, es decir, el cliente y el resto de agentes que intervienen en el proceso. Su estructura es:

```
  <partnerLinks>
    <partnerLink name="assessor"
     partnerLinkType="ns2:AssessorService1"
     partnerRole="AssessorServicePortTypeRole"/>

    <partnerLink name="approver"
     partnerLinkType="ns1:ApprovalService1"
     partnerRole="ApprovalServicePortTypeRole"/>

    <partnerLink myRole="LoanServicePortTypeRole"
     name="client"
     partnerLinkType="ns3:LoanService1"/>
  </partnerLinks>
```

3. La tercera parte se encarga de definir las variables que se van a utilizar a lo largo de todo el proceso. Su estructura es:

```
<variables>
 <variable messageType="ns3:LoanServiceOperationReply"
  name="processOutput"/>
 <variable messageType="ns3:LoanServiceOperationRequest"
  name="processInput"/>
 <variable messageType="ns2:AssessorServiceOperationReply"
  name="assessorOutput"/>
 <variable messageType="ns2:AssessorServiceOperationRequest"
```

```
  name="assessorInput"/>
 <variable messageType="ns1:ApprovalServiceOperationReply"
  name="approverOutput"/>
 <variable messageType="ns1:ApprovalServiceOperationRequest"
  name="approverInput"/>
</variables>
```

4. La cuarta parte está formada por todos los manejadores de eventos que tiene el proceso. Estos manejadores pueden ser de fallos, de casos excepcionales en los cuales se hace una petición particular, etcétera. En este caso, LoanApproval no tiene ninguno, pero, el esquema genérico de un manejador de eventos es el siguiente:

```
<bpel:faultHandlers>
 <bpel:catchAll>
  <bpel:sequence name="name_sequence">

   <bpel:assign name="name_assign">
    <bpel:copy>
     <bpel:from>'Cadena a copiar'</bpel:from>
      <bpel:to part="name_part" variable="name_variable">
        <bpel:query>client:error</bpel:query>
      </bpel:to>
     </bpel:copy>
   </bpel:assign>

   <bpel:reply faultName="client:ManejadorError"
     name="name_reply" operation="name_operation"
     partnerLink="client" portType="client:Manejador"
     variable="ManejadorGeneral"/>

  </bpel:sequence>
 </bpel:catchAll>
</bpel:faultHandlers>
```

Tiene dos partes fundamentales:

a) Assign: nos indica qué variables hay que completar o qué mensajes debemos dar cuando se produce ese evento en particular.

b) Reply: nos indica dónde debemos responder o enviar el resultado de manejar el evento.

5. Y la última parte se corresponde con el proceso en sí. En él se describen todas las actividades que se realizan en el proceso, a través de las actividades que proporciona el lenguaje. Un ejemplo de esta última parte puede ser la siguiente:

```
<sequence name="Main">
  <receive createInstance="yes" name="Receive1"
   operation="grantLoan" partnerLink="client"
   portType="ns3:LoanServicePortType"
   variable="processInput"/>
  <assign name="DefaultValues">
    <copy>
      <from>
        <literal><ns0:AssessorRequest><ns0:amount>
         0.0
        </ns0:amount></ns0:AssessorRequest></literal>
      </from>
      <to part="input" variable="assessorInput"/>
    </copy>
    ...
  </assign>
  <if name="If1">
    <condition>
     (number(string($processInput.input/ns0:amount))<= 10000)
    </condition>
    <sequence name="SmallAmount">
      <assign name="copyLoanInfoToAssessorInput">
        <copy>
          <from>$processInput.input/ns0:amount</from>
          <to>$assessorInput.input/ns0:amount</to>
        </copy>
      </assign>
      <invoke inputVariable="assessorInput"
        name="queryAssessor"
        operation="assessLoan"
        outputVariable="assessorOutput"
        partnerLink="assessor"
        portType="ns2:AssessorServicePortType"/>
      ...
    </sequence>
    <else>
      ...
    </else>
  </if>
  <reply name="Reply1" operation="grantLoan"
    partnerLink="client"
    portType="ns3:LoanServicePortType"
    variable="processOutput"/>
</sequence>
```

Se caracteriza porque todas las actividades del proceso se encuentran recogidas en la actividad principal “sequence”. Las actividades más importantes, algunas de ellas utlizadas en este último ejemplo, son:

- Assign: al igual que en manejador de eventos, nos permite asignar valores a ciertas variables.
- If: condicional que podemos encontrarnos dentro del proceso. Está formado por una condición que según se cumpla o no, hará que se ejecute la secuencia siguiente Puede encontrase sólo, o acompañado de “Else”.
- Else: complemento al condicional “If” que indica lo que hacer en caso contrario.
- Receive: permite al proceso recibir todos los mensajes de entrada desde los agentes correspondientes.
- Reply: devuelve la respuesta al mensaje recibido anteriormente por la actividad “Receive”.
- Flow: actividad que nos permite ejecutar de forma concurrente todas las actividades que recoja.
- While: permite la iteración de las actividades recogidas por ellas, mientras se cumpla la condición establecida.
- ForEach: tiene la misma funcionalidad que la actividad “While”, pero, las iteraciones se producen un número determinado de veces, dando la ventaja de que nunca pueda producirse un bucle infinito debido a la condición establecida.
- Invoke: invoca un servicio web en un lugar necesario del proceso.
- CorrelationSets: permite la vinculación de un mensaje con una instancia en particular.
- Correlations: inicializa una de las instancias que se ha referenciado anteriormente con la actividad CorrelationSets.
- Scope: nos permite declarar recursos locales en una actividad en particular. Estos recursos locales pueden ser variables, manejadores de eventos, agentes externos ...

3.3. BPTS, BPELUnit Test Suite

Uno de las partes más importantes para el desarrollo de este Proyecto Fin de Carrera ha sido el estudio de los ficheros BPTS, o lo que es lo mismo, BPEL Unit Test Suite [11]. Como ya se comentó en la Introducción, se trata de un conjunto de casos de prueba para BPELUnit recogidos en un fichero XML que define qué proceso se va a probar y cómo.

Al trabajar con cada una de las composiciones, hemos tenido en nuestra mano un BPTS de cada una, con una serie de casos de prueba específicos. Pues bien, para comprender un poco mejor la organización de estos ficheros, vamos a explicar brevemente la estructura genérica de uno de ellos, basándonos en el BPTS correspondiente de la composición LoanApproval:

1. En primer lugar, hay una parte previa en la cual se encuentran las conexiones con la URL correspondiente y con los servicios WSDL. Su estructura es la siguiente:

```
<tes:name>LoanServiceTest</tes:name>
<tes:baseURL>http://localhost:7777/ws</tes:baseURL>

<tes:deployment>
 <tes:put name="LoanApprovalProcess" type="activebpel">
   <tes:wsdl>LoanService.wsdl</tes:wsdl>
   <tes:property name="BPRFile">
     LoanApprovalDoc.bpr
   </tes:property>
 </tes:put>
 <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
 <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
</tes:deployment>
```

2. Y una segunda parte, que es la que se corresponde con todos los casos de prueba correspondientes. Se encuentran recogidos en la actividad “TestCases” y puede haber desde una sola actividad, hasta varias. Cada caso de prueba se encuentra recogido en la actividad “TestCase”. Su estructura es la siguiente:

```
<tes:testCases>
 <tes:testCase name="LargeAmount"
  basedOn="" abstract="false"
  vary="false">

  <tes:clientTrack>
   <tes:sendReceive
    service="sp:LoanServiceService"
    port="LoanServicePort"
    operation="grantLoan">

    <tes:send fault="false">
     <tes:data>
      <esq:ApprovalRequest>
       <esq:amount>150000</esq:amount>
      </esq:ApprovalRequest>
     </tes:data>
    </tes:send>

    <tes:receive fault="false">
     <tes:condition>
      <tes:expression>
```

```
          esq:ApprovalResponse/esq:accept
        </tes:expression>
         <tes:value>'true'</tes:value>
        </tes:condition>
      </tes:receive>
     </tes:sendReceive>
    </tes:clientTrack>

    <tes:partnerTrack name="approver">
     <tes:receiveSend
      service="ap:ApprovalServiceService"
      port="ApprovalServicePort"
      operation="approveLoan">

      <tes:send fault="false">
       <tes:data>
        <esq:ApprovalResponse>
         <esq:accept>true</esq:accept>
        </esq:ApprovalResponse>
       </tes:data>
      </tes:send>

      <tes:receive fault="false"/>
     </tes:receiveSend>
    </tes:partnerTrack>

    <tes:partnerTrack name="assessor"/>

   </tes:testCase>

   <tes:testCase name="LargeAmountRejected"
     basedOn="" abstract="false" vary="false">
   ...
   </tes:testCase>
    ...
</tes:testCases>
```

Cada caso de prueba recoge una serie de actividades que tienen una función determinada. Estas actividades son:

- ClientTrack: define la petición que se enviará desde el cliente simulado. Sólamente puede haber un cliente por caso de prueba.
- SendReceive: recoge la entrada y salida de mensajes que realiza el agente que la gestiona.
- Send: envía un mensaje con una petición síncrona.

- Receive: evaluará su contenido cuando el proceso responda al envío realizado por el cliente.
- PartnerTrack: define un "mockup" o servicio, y está compuesto por las mismas actividades que el "ClientTrack". Puede haber más de uno dentro del mismo caso de prueba y cada uno con sus actividades y funciones correspondientes.

Como podemos ver, se trata de un código bastante engorroso y complejo. Por ello, para facilitar el estudio de las diversas composiciones, se comenzó a desarrollar los ficheros BPTS, pero, basados en plantillas CSV. Basicamente, consiste en generalizar una serie de casos de prueba en uno solo, y guardar en un fichero CSV los distintos valores relevantes de cada caso. En el siguiente apartado explicaremos cuál es la estructura de un fichero CSV y las modificaciones que implica su uso en los ficheros BPTS.

3.4. CSV, Comma-Separated Values

Los ficheros CSV (Comma-Separated Values) son un tipo de documento de formato sencillo que nos permite representar datos en forma de tabla. Las columnas son separadas por comas y las filas por saltos de línea. El formato de estos archivos no requiere una codificación especial ni un formato de línea concreto.

Para ver con un poco más de claridad la organización de estos ficheros, vamos a describir a continuación los puntos más importantes que deben seguir:

1. Para separar cada una de las celdas utilizamos la coma (,). En los países cuyo separador decimal es la coma, utiliza otro signo de sepación como el punto y coma (;).

2. Si en alguna de las celdas queremos escribir un carácter especial como una coma o un salto de línea, debemos escribir el contenido de esa celda entre comillas dobles ("carácter especial").

3. La primera fila de un fichero CSV contiene el nombre de cada una de las columnas, indicando así qué es lo que contiene cada una.

```
req_amount,ap_reply,ap_limit,as_reply,as_limit,accepted
1500,true,0,high,0,true
1500,false,0,high,0,false
20000,false,0,low,0,false
4000,true,0,low,0,true
```

Cada una de las líneas, a partir de la segunda, representa un caso de prueba. Por tanto, lo que queremos conseguir es que cada composición pueda guardar en un fichero CSV todos y cada uno de sus casos de prueba, y probarlos con un fichero BPTS que esté preparado para trabajar con plantillas CSV. De forma que, para poder realizar esto, debemos modificar levemente el

fichero BPTS original que teníamos, para que a la hora de ejecutarlos, tome los valores correspondientes del fichero CSV.

Para comenzar con esta tarea, disponemos de un BPTS basado en plantillas que fue realizado por uno de los componentes del grupo de investigación UCASE, Antonio García Domínguez. Es el BPTS basado en plantillas de la composición LoanApproval, el cual utiliza un fichero CSV llamado "data.csv". Para comprender un poco mejor cómo vamos a trabajar más adelante para obtener el resto de BPTS de las otras composiciones, vamos a explicar un poco las transformaciones que se producen en el BPTS y cómo utiliza los datos del fichero CSV dentro del fichero LoanApprovalProcess-CSV.bpts.

Los cambios más relevantes entre ambos tipos de BPTS son:

- En primer lugar, pasamos de tener varios casos de prueba a sólamente uno. Y dentro de él, se encuentran distribuídas las variables que van a ser sustituídas por los valores que se encuentren en el fichero CSV.

```
<tes:testCases>
 <tes:testCase name="MainTemplate" basedOn=""
  abstract="false" vary="false">
  ...
  <tes:clientTrack>
   <tes:sendReceive
    service="sp:LoanServiceService"
    port="LoanServicePort"
    operation="grantLoan">

    <tes:send fault="false">
     <tes:template>
      <esq:ApprovalRequest>
       <esq:amount> $req_amount</esq:amount>
      </esq:ApprovalRequest>
     </tes:template>
    </tes:send>
   ...
 </tes:testCase>
</tes:testCases>
```

  Como podemos ver, el valor numérico que aparecía anteriormente, pasa a ser $req_amount que nos permitirá sustituir en él los valores que tenga el fichero data.csv en su columna. Otro cambio importante, es que en lugar de obtener los datos de entrada dentro de la actividad <tes:data>, ahora se utiliza <tes:template>para indicar que lo obtenemos desde una plantilla.

- Otro cambio importante, que más que cambio se le puede llamar novedad, es la introducción de la actividad <tes:setUp>y en su interior, <tes:dataSource>. Nos permite indicarle

al fichero BPTS cuál va a ser el fichero donde se encuentren los datos y cómo interpretarlos. Se añade dentro del caso de prueba, justo antes de todas las actividades relacionadas con la composición:

```
<tes:setUp>
 <tes:dataSource type="csv" src="data.csv">
  <tes:property name="separator">,</tes:property>
 </tes:dataSource>
</tes:setUp>
```

Con <tes:dataSource>le indicamos al BPTS el tipo de fichero con el que se va a encontrar, en este caso CSV, y el nombre del fichero donde se encuentran los datos. Y con <tes:property>le decimos que los datos del fichero se van a encontrar separados por el símbolo ",".

Además de estas particularidades, el código que guardan los ficheros BPTS basados en plantillas puede llevar también código Velocity. En este ejemplo que estamos explicando, hay una parte en la cual se elige mediante un condicional que lleva una expresión en XPath, si se toma un valor u otro:

```
<tes:send fault="false">
 <tes:template>
  <esq:ApprovalResponse>
   #set($amount = $integer.parseInt($xpath.evaluateAsString("//esq:amount",$request)))
   #if($as_reply != 'smart' )
     <esq:accept>$ap_reply</esq:accept>
   #elseif($integer.parseInt($ap_limit) > $amount )
     <esq:accept>true</esq:accept>
   #else
     <esq:accept>false</esq:accept>
   #end
  </esq:ApprovalResponse>
 </tes:template>
</tes:send>
```

A la hora de realizar los nuevos BPTS de las otras composiciones, esto no será necesario, ya que con las actividades que nos proporciona el lenguaje y la lógica de la composición, no será necesario utilizarlo.

3.5. Servicios Web

Los servicios web son sistemas software que utilizan un conjunto de protocolos y estándares para permitir la interacción máquina-a-máquina sobre una red. Su interfaz está descrita en WSDL y el resto de sistemas pueden comunicarse con él a través de mensajes SOAP. Se pueden utilizar estos servicios web para intercambiar datos en redes de ordenadores [10].

Con ellos se definen un nuevo estilo de arquitectura abstracta que no está limitado a ninguna tecnología en particular. Son una implementación del paradigma SOA, dentro del cual destacan tres figuras importantes:

1. Proveedor de servicios: es el encargado de crear el servicio web, publicar su interfaz y la información de acceso al registro agente del servicio.

2. Agente de servicios: es el encargado de construir la interfaz del servicio web y de la implementación accesible a cualquier solicitante del servicio.

3. Consumidor de servicios: es el solicitante del servicio web y se une al proveedor de servicios para invocar a uno de sus servicios web.

Juegan un papel muy importante dentro de este Proyecto Fin de Carrera, ya que las composiciones WS-BPEL se encargan de invocar a los servicios web y son los que nos proporcionan parte de los resultados que estamos buscando [20].

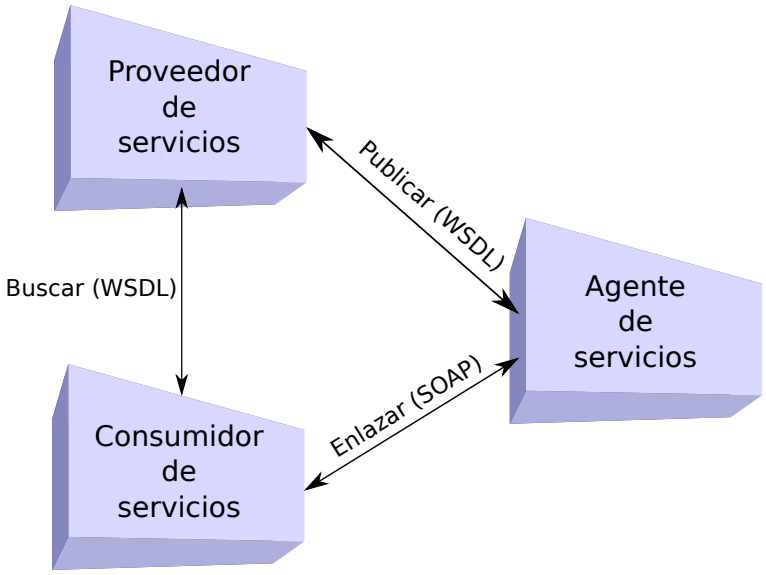

Figura 3.1: Arquitectura de los Servicios Web

3.6. Prueba de mutaciones

La prueba de mutaciones es una técnica de prueba de software de caja blanca, basada en los posibles errores que pueden cometer los programadores a la hora de desarrollar un programa. Las pruebas de software de caja blanca están centradas en la funcionalidad del software, implicando una gran vinculación con el código fuente. La persona encargada de llevarlas a cabo escoge diversos valores de entrada para probar todos los caminos posibles de ejecución, y de esta forma, cerciorarse de que los valores de salida devueltos son los adecuados.

Durante las pruebas de mutaciones, se introducen pequeños fallos en el programa original que vamos a estudiar con los operadores de mutación, generando así varios programas con fallos (mutantes) que son a los que les aplicaremos los distintos casos de prueba que disponemos. Uno de los problemas de esta técnica es que conlleva un gran coste computacional. Debido a que normalmente, dispone de una gran cantidad de operadores de mutación que generan un amplio número de mutantes. Y estos a su vez, son ejecutados sobre el conjunto de casos de prueba, hasta que mueran.

El proceso de prueba de mutaciones comienza con el análisis de mutaciones. Es el proceso que mide la calidad del conjunto de casos de prueba [14]. En dicho proceso, como ya hemos comentado anteriormente, se generan los programas que contienen uno o más fallos respecto al original (mutantes). Esto se consigue al aplicar los operadores de mutación, que introducen pequeños cambios sintácticos que están basados en los errores más usuales que suelen cometer los programadores. De esta forma, forzamos a probar la corrección del programa, a partir de cambios que mantienen su validez sintáctica.

En la Figura 3.2 podemos ver de forma más clara cuál es el esquema general del proceso de análisis de mutación [15].

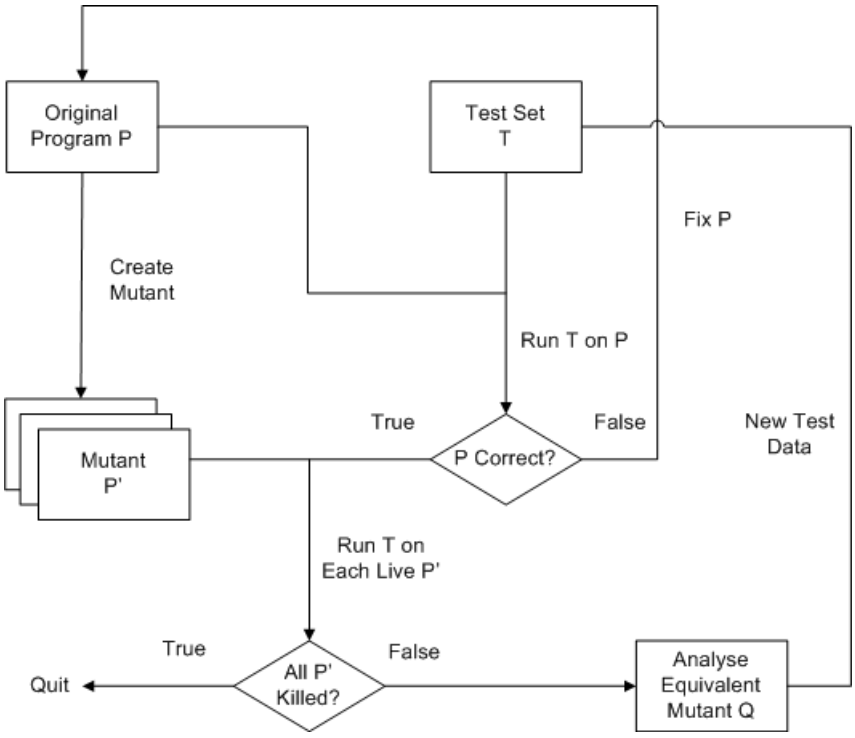

Figura 3.2: Proceso de análisis de mutaciones

Podemos distinguir dos tipos claros de mutantes, en función del número de cambios sintácticos introducidos:

- Mutantes de primer orden: son aquellos que inlcuyen sólo un cambio respecto del pro-

grama original. Un ejemplo puede ser éste:
Programa original

```
if(x < 10)...
```

Mutante de primer orden

```
if(x > 10)...
```

- Mutantes de orden superior: son aquellos que inlcuyen más de un cambio respecto del programa original. Un ejemplo puede ser éste:
  Programa original

```
if(x <10)...
```

  Mutante de orden superior

```
if(x >100)...
```

Una vez que ya hemos generado los mutantes, probamos el conjunto de casos de prueba con ellos. Y según el resultado que obtengamos, podemos diferenciar tres tipos distintos de mutantes:

1. Mutante muerto: la salida que obtenemos del mutante es distinta a la del programa original en uno o varios casos de prueba.
2. Mutante vivo: la salida que obtenemos del mutante es igual a la del programa original en todos los casos de prueba. Podemos hacer una subdivisión en este tipo de mutantes:
   a) Equivalente: la salida que obtenemos del mutante es siempre la misma que la del programa original para todos los casos de prueba.
   b) Resistente: el conjunto de casos de prueba no es suficiente como para detectar los fallos.
3. Mutante erróneo: el mutante produce algún error al probarlo y no puede ser ejecutado.

Después de probar los mutantes, se pasa a calcular la calidad del conjunto de casos de prueba mediante *la puntuación de mutación (mutation score)*. Esto es, el cociente entre el número de mutantes muertos y el número de mutantes no equivalentes. Por tanto, para poder calcular esta puntuación, debemos detectar cuáles de nuestros mutantes son equivalentes. Ésta es una de las dificultades que presenta esta técnica, ya que es complicado detectarlos. Un dato muy importante para clasificar un mutante como equivalente o no, es saber si nuestro conjunto de casos de prueba es suficiente como para poder diferenciar los mutantes.

3.7. MuBPEL

MuBPEL es uno de los pilares principales de nuestro estudio, es una herramienta de pruebas de mutación de Web Services Business Process Execution Language (WS-BPEL) 2.0. Puede ser utilizado para evaluar la calidad de un conjunto de pruebas, comprobando si se puede diferenciar la salida de un mutante de la salida del programa original. Una vez que tengamos todos los mutantes que vamos a estudiar, pasaremos a probar todos los casos de prueba que generaremos a partir de las relaciones metamórficas que vamos a desarrollar de cada una de las composiciones. En capítulos posteriores, se encuentran los manuales de instalación y de usuario para saber cómo utilizar esta herramienta.

Su estudio, junto a los conocimientos de las Pruebas Metamórficas, fue una parte fundamental a la hora de comenzar con el Proyecto Fin de Carrera, ya que era la forma de saber cómo un mutante era matado o no por un caso de prueba. Nos podemos encontrar con tres posibles casos a la hora de aplicar un caso de prueba en un mutante:

1. Salida 0: Se producen las mismas salidas a la hora de comparar el programa original con el programa mutante.

2. Salida 1: Se producen salidas distintas a la hora de comparar el programa original con el programa mutante.

3. Salida 2: Se ha producido algún error en la ejecución.

De forma, que si tenemos varios casos de prueba que aplicar a varios mutantes, podemos encontrarnos con la siguiente situación:

```
m17-04-01.bpel 1 0 1 0 1 0 0 1 0
m17-05-01.bpel 1 0 0 0 1 1 1 1 0
m17-06-01.bpel 0 0 0 0 0 0 0 0 0
m26-01-01.bpel 2 2 2 2 2 2 2 2 2
```

Cada fila tiene un nombre al principio, que representa qué mutante se está estudiando, y el resto de números que completa la fila, representan el resultado de aplicar cada caso de prueba que diponemos en el BPTS a dicho mutante. Pues bien, si le echamos un vistazo a estos mutantes que estamos estudiando, podemos encontrarnos tres casos distintos:

1. Mutante vivo: Es el caso del mutante m17-06-01.bpel, todos los casos de prueba que se le han aplicado han tenido la misma salida que el programa original.

2. Mutante muerto: Es el caso del mutante m17-04-01.bpel y m17-05-01.bpel, alguno de los casos de prueba que se le han aplicado ha obtenido una salida distinta a la del programa original, por tanto, es un fallo que se ha detectado.

3. Mutante erróneo: Es el caso del mutante m26-01-01.bpel, todos sus casos de prueba tienen como salida 2 debido a un error en su ejecución.

Nuestro objetivo es conseguir matar los mutantes que queden vivos y así poder detectar posibles fallos que anteriormente no se hayan encontrado. Por tanto, nuestras relaciones metamórficas deben estar relacionadas con los mutantes. Esto quiere decir, que debemos encontrar cuál es la diferencia entre el programa original y el mutante en sí, porque de esta forma, podemos encontrar alguna pista de cómo desarrollar la relación metamórfica y encontrar el fallo.

4 Composiciones de servicios web utilizadas

En este Proyecto Fin de Carrera, como ya hemos comentado anteriormente, nos vamos a centrar en el estudio de tres composiciones: LoanApproval, MarketPlace y MetaSearch. Cada una tiene una lógica distinta, lo que nos permite desarrollar diferentes relaciones metamórficas, aunque, hay relaciones que pueden ser aplicadas en varias composiciones, lo que nos indica que existe algo en común entre ellas. A continuación, vamos a explicar detalladamente cómo se comporta cada una de las composiciones y de qué recursos partimos para el estudio de cada una. De esta forma, sabremos cuáles son las partes que tenemos que desarrollar de cada composición.

4.1. LoanApproval

LoanApproval, o lo que es lo mismo, *Gestión de Préstamos*, tiene una lógica bastante sencilla. Básicamente, consiste en un proceso de aprobación de préstamos a partir de una cantidad monetaria y de la opinión de un aprobador de préstamos y un asesor financiero.

Existe un límite en la cantidad de dinero del préstamo ($req_amount), en nuestro caso 10000$, que según si se supera o no, entrará en acción un agente u otro. Pueden darse dos casos:

- La cantidad es inferior o igual a 10000, entonces, se llamará al asesor para que nos indique si existe un riesgo a la hora de concedernos el préstamo. De nuevo nos podemos encontrar con dos posibles opciones:
  - Si su respuesta es “low” ($as_reply), es decir, que existe muy poco riesgo a la hora de concedernos el préstamo, directamente se nos concederá y obtendremos por respuesta “true” ($accepted).
  - Si su respuesta es “high” ($as_reply), es decir, que existe riesgo a la hora de concedernos el préstamo, pasará a trabajar el aprobador. Es el agente que tendrá la última palabra en la respuesta al cliente. Si su respuesta es “true” ($ap_reply), se nos concederá el préstamo y el cliente obtendrá por respuesta “true” ($accepted). En caso contrario, si la respuesta del aprobador es “false” ($ap_reply), no se nos concederá el préstamo y el cliente obtendrá por respuesta “false” ($accepted).
- La cantidad es mayor a 10000, entonces, se llama directamente al aprobador (en este caso el asesor no juega ningún papel) y es el que decidirá si se concede o no el préstamo. Si

su respuesta es “true” ($ap_reply), se nos concederá el préstamo y el cliente obtendrá por respuesta “true” ($accepted). En caso contrario, si la respuesta es “false” ($ap_reply), no se nos concederá el préstamo y el cliente obtendrá por respuesta “false” ($accepted).

Para comprender un poco mejor cómo funciona esta lógica, en el siguiente gráfico puede verse claramente cuáles son los diversos caminos que pueden darse en esta composición [5]:

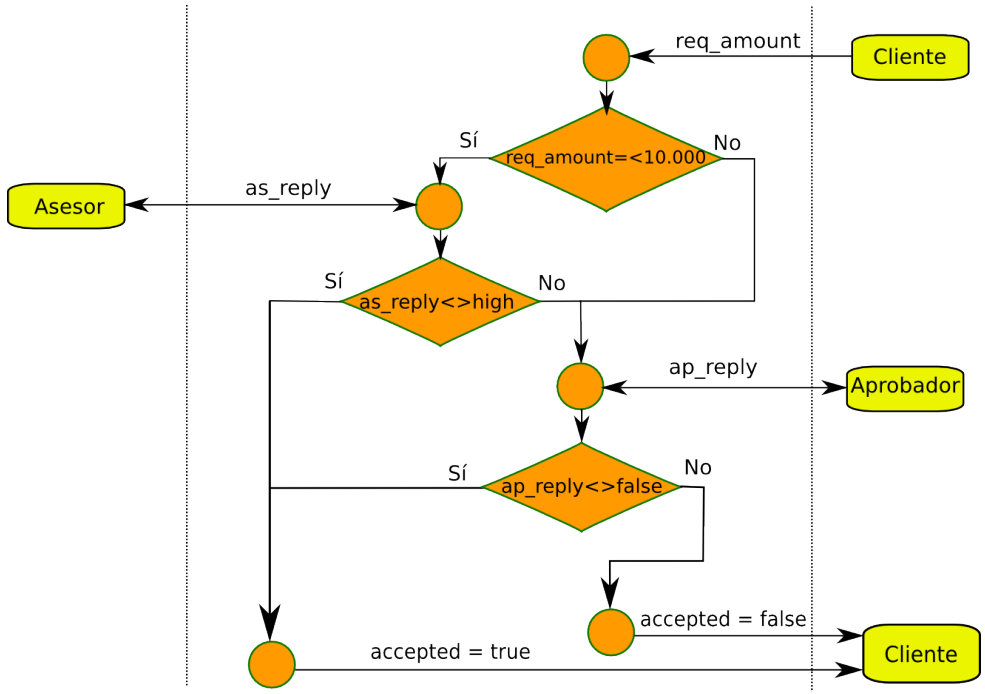

Figura 4.1: Lógica de la composición LoanApproval

4.1.1. Recursos y objetivos por cumplir

Como recursos de partida para su estudio disponemos de los siguientes ficheros: LoanApprovalProcess.bpel, LoanApprovalProcess-CSV.bpts, LoanApprovalProcess-Velocity.bpts, data.csv y los correspondientes ficheros wsdl que implementan la interfaz de los servicios web (ApprovalService.wsdl, AssessorService.wsdl, LoanService.wsdl).

Es decir, que uno de los objetivos que queremos conseguir con cada composición está ya hecho: BPTS basado en plantillas CSV. Por tanto, en este caso debemos encontrar cuáles son los mutantes que aún quedan vivos después de aplicarles los casos de prueba que ya tenemos, y a partir de ahí desarrollar las relaciones metamórficas correspondientes para generar los casos de prueba que puedan matarlos, es decir, que detecten el error existente en la composición.

Así que otro objetivo pendiente es generar relaciones metamórficas basadas en condiciones aritméticas y lógicas relacionadas con los mutantes que queden vivos una vez que hayamos hecho las pruebas correspondientes entre los mutantes y los casos de prueba con MuBPEL.

Una vez que las tengamos, desarrollar algún caso de prueba "a mano" basado en ellas que nos mate alguno de ellos. Por último, el otro objetivo pendiente que nos queda es automatizar esta generación de casos de prueba. Es decir, trasladar a código Java las relaciones metamórficas que desarrollemos y así obtener más casos de prueba.

Una vez que ya hayamos conseguido todos estos objetivos, lo que nos quedará será realizar pruebas en los mutantes vivos con todos los casos de prueba que hayamos generado gracias a la automatización. Y comprobar si hemos obtenido resultados relevantes para la investigación en las Pruebas Metamórficas.

4.2. MarketPlace

La composición de MarketPlace, Mercado de compraventa, es la más sencilla de todas las que vamos a estudiar. Tiene una lógica muy simple, ya que básicamente se centra todo en un simple condicional. Las novedades que presenta frente a LoanApproval, es que esta introduce la concurrencia a través de la actividad "Flow", y la aparición de la actividad "delaySequence" que permite controlar el tiempo de respuesta de cada agente dentro de cada caso de prueba.

Existen dos agentes principales: el vendedor, que es el que pone el producto en el mercado con un precio de venta inicial; y el comprador, que es el que ofrece un precio por el producto, que debe de ser mayor o igual al que ha propuesto el vendedor para poder comprarlo. Es una especie de puja, pero, con un solo comprador. Pues bien, la composición funciona de la siguiente manera:

- Por cada caso de prueba, se pone en concurrencia dos iteraciones suyas y en cada una, cada agente tiene un tiempo de respuesta que viene dado por la actividad "delaySequence". Todas las iteraciones de un caso de prueba ejecutan lo mismo, sólo que pueden tener un retardo a la hora de dar su respuesta.
- Una vez que nos encontramos dentro del caso de prueba, ocurre lo siguiente:
  - El vendedor ofrece su producto ($Sel_Item) y un precio inicial de venta ($Sel_Price).
  - El comprador da un precio de compra ($Buy_Price) por el producto ($Buy_Item) ofrecido por el vendedor.
  - Si el precio del comprador es mayor o igual que el ofrecido por el vendedor ($Buy_Price >= $Sel_Price), se producirá la compra y obtendremos como salida para ambos agentes "Deal Successful" ($Sel_Value, $Buy_Value).
  - Si en caso contrario, el precio del comprador ($Buy_Price) no supera al precio del vendedor ($Sel_Price), no se producirá la compra y obtendremos como salida para ambos agentes "Deal Failed" ($Sel_Value, $Buy_Value).

Para ver más claramente cómo se organiza esta lógica, en el siguiente gráfico tenemos representada la composición, incluyendo la concurrencia entre los distintos casos de prueba:

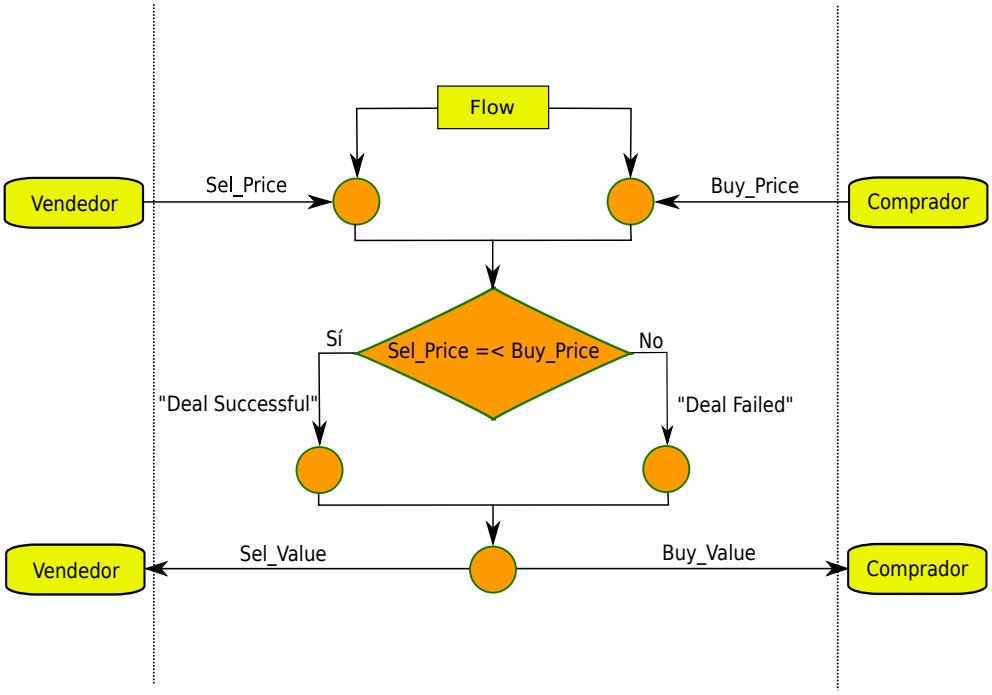

Figura 4.2: Lógica de la composición MarketPlace

4.2.1. Recursos y objetivos por cumplir

Como recursos de partida para su estudio disponemos de los siguientes ficheros: marketplace.bpel, marketplace.bpts y marketplace.wsdl que se encarga de implementar la interfaz de los servicios web que se invoquen en la composición. En este caso, no disponemos de un fichero BPTS basado en plantillas que esté ya preparado para trabajar con archivos CSV donde guardemos los diferentes casos de prueba.

Por tanto, los objetivos que aún nos quedan pendientes son:

1. Desarrollar un fichero BPTS basado en plantillas de la composición, el cual esté preparado para trabajar con ficheros CSV.

2. Trasladar todos los casos de prueba que tenemos en el BPTS original a un fichero CSV.

3. Comprobar si realmente la plantilla que hemos obtenido junto al fichero CSV reflejan completamente las pruebas de la composición.

4. Encontrar cuáles son los mutantes vivos después de aplicar el conjunto de casos de prueba original.

5. Diseñar las relaciones metamórficas correspondientes para matar esos mutantes que aún queden vivos.

6. Obtener algún caso de prueba "a mano" a partir de las relaciones metamórficas que hemos desarrollado anteriormente que nos demuestre que realmente la relación metamórfica es efectiva.

7. Automatizar todas las relaciones metamórficas que hemos diseñado anteriormente y así obtener un mayor número de casos de prueba.

Utilizaremos MuBPEL para comprobar la respuesta de los casos de prueba aplicados a los mutantes, y estudiaremos las diferencias existentes entre los mutantes que aún siguen vivos y el fichero BPEL original. A partir de estas diferencias, desarrollaremos las relaciones metamórficas necesarias y buscaremos algún caso de prueba que sea relevante a la hora de aplicarlo a los mutantes vivos.

Automatizaremos las relaciones metamórficas que diseñemos, y una vez que ya hayamos conseguido todos los objetivos, lo que nos quedará será realizar pruebas en los mutantes vivos con todos los casos de prueba que hayamos generado gracias a la automatización. Y comprobar si hemos obtenido resultados relevantes para la investigación en las Pruebas Metamórficas.

4.3. MetaSearch

La última composición que vamos a estudiar va a ser la llamada MetaSearch, o también conocida como Meta Búsqueda. Al contrario que la anterior, es la composición más complicada, ya que mezcla concurrencia, control temporal con "delaySequence", manejadores de eventos, mezcla de concurrencia dentro de fragmentos secuenciales, variables globales y locales (anteriormente sólo hemos trabajado con variables globales), y además, una larga extensión. Su lógica no es complicada en sí, pero, la comprensión del código es lo que dificulta su estudio.

Podemos encontrar tres agentes distintos en esta composición: el cliente, que es el que realiza la consulta de búsqueda; Google, que es uno de los motores de búsqueda; y MSN, que es el segundo motor de búsqueda. Básicamente consiste en que un cliente, o usuario más bien, realiza una consulta utilizando dos motores de búsqueda. Tiene algunos matices que vamos a explicar a continuación:

- El usuario introduce unos datos iniciales para que puedan comenzar la búsqueda los respectivos motores ($Query, $Lenguaje, $Country, $Max_Results).

- Cada agente (Google y MSN) comienza la búsqueda en paralelo, pudiendo obtener cada uno unos resultados independientes del otro.

- Si entre los dos motores de búsqueda se obtiene algún resultado ($Results >0), pasamos a mostráselos al cliente:
    - Si existe algún resultado de Google ($Google_Results >0), se mostrarán al usuario primero todos los resultados de Google ($url1, $title1, $snippet1, $url2, $title2, $snippet2, $url3, $title3, $snippet3).

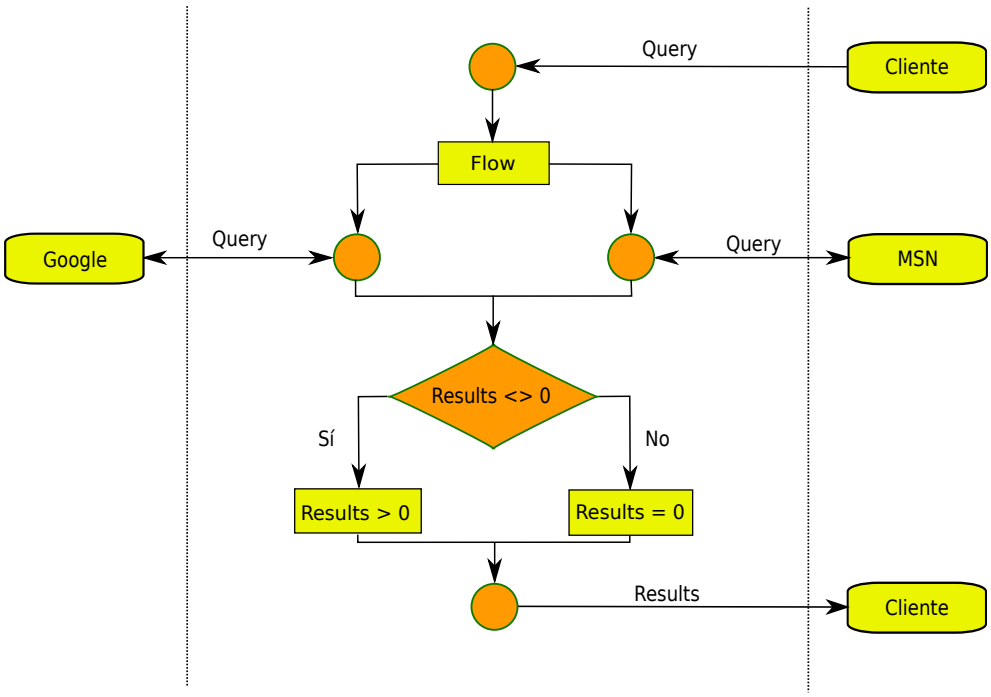

Figura 4.3: Lógica de la composición MetaSearch

- Si existe algún resultado de MSN ($Results - $Google_Results >0), se mostrarán al usuario, después de los posibles resultados de Google, todos los resultados de MSN ($title_msn1, $description1, $url_msn1, $title_msn2, $description2, $url_msn2, $title_msn3, $description3, $url_msn3).
- Si no hubiera resultados por parte del motor de búsqueda Google ($Google_Results = 0), sólo se mostrarían los resultados de MSN.
- Si no hubiera resultados por parte del motor de búsqueda MSN ($Results - $Google_Results = 0), sólo se mostrarían los resultados de Google.

En la Figura 4.3 podemos ver mejor cómo es la lógica de la composición. Las condiciones del orden de salida de los resultados no están reflejadas, pero, siguen siendo las mismas que acabamos de comentar.

4.3.1. Recursos y objetivos por cumplir

Para comenzar con el estudio de esta composición disponemos de los siguientes ficheros: MetaSearchBPEL2_forEach.bpel, MetaSearchTest.bpts, MetaSearchTest-Extended-Lorena.bpts y los ficheros que se encargan de implementar los servicios web que invocamos a lo largo de la composición (GoogleBridge.wsdl, MetaSearch.wsdl, msnsearch.wsdl, msnsearchRef.wsdl).

En este caso, disponemos de dos ficheros BPTS con casos de pruebas, pero, ninguno está basado en plantillas para ficheros CSV. El fichero MetaSearchTest.bpts son los casos de prueba originales de la composición, y el fichero MetaSearchTest-Extended-Lorena.bpts contiene

casos de prueba desarrollados por Lorena Gutiérrez Madroñal, que pertenece al grupo de investigación UCASE y su estudio también está relacionado con esta composición.

Prácticamente, tenemos pendiente el mismo trabajo que en MarketPlace, sólo que éste es más extenso. Por tanto, los objetivos que debemos completar aún son:

1. Desarrollar un fichero BPTS basado en plantillas de la composición, el cual esté preparado para trabajar con ficheros CSV.

2. Trasladar todos los casos de prueba que tenemos en el BPTS original y en el extendido a un fichero CSV.

3. Comprobar si realmente la plantilla que hemos obtenido junto al fichero CSV reflejan completamente las pruebas de la composición.

4. Encontrar cuáles son los mutantes vivos después de aplicar el conjunto de casos de prueba original.

5. Desarrollar las relaciones metamórficas correspondientes para matar esos mutantes que aún queden vivos.

6. Obtener algún caso de prueba "a mano" a partir de las relaciones metamórficas que hemos desarrollado anteriormente que nos demuestre que realmente la relación metamórfica es efectiva.

7. Automatizar todas las relaciones metamórficas que hemos desarrollado anteriormente y así obtener un mayor número de casos de prueba.

De forma que intentaremos solventar todos los problemas que nos vayan surgiendo a lo largo de su estudio e intentaremos obtener los resultados más relevantes posibles para el posterior estudio completo de la composición. Éste trabajo futuro se explicará mejor en capítulos posteriores.

5 Estudios con MuBPEL

En este capítulo vamos a centrarnos en el estudio previo con MuBPEL de cada una de las composiciones, desde el número de mutantes que se generan, las salidas que nos ofrece cada uno de ellos y los mutantes en los que nos vamos a centrar en cada composición. Como ya hemos comentado anteriormente en el capítulo de "Fundamentos", podemos distinguir tres tipos de mutantes: vivos, muertos y equivalentes. Pues bien, haremos una clasificación de los mutantes y explicaremos cuáles van a ser los que intentaremos detectar en cada caso.

Para realizar todas estas pruebas hay que seguir los pasos que se describen en el apéndice "Manual de usuario", en el cual se explica cómo trabajar con las composiciones en MuBPEL. Vamos a dividir la explicación en tres partes, una por cada composición, para así evitar confusiones y poder explicar más claramente los mutantes de cada composición, y las diferencias existentes entre los vivos y el fichero BPEL original.

5.1. El problema del operador "//"

Antes de comenzar con el estudio de cada composición, me gustaría explicar uno de los muchos mutantes que nos vamos a encontrar. Es el caso de los mutantes cuya diferencia con el original es un simple "//" en lugar de un "/". Se puede dar en casos como este:

- Original:

```
<bpel:query>ns2:Request /ns2:AppID</bpel:query>
```

- Mutante:

```
<bpel:query>ns2:Request //ns2:AppID</bpel:query>
```

La diferencia entre ellos es que "ns2:Request/ns2:AppID" busca a los hijos "ns2:AppID" de "ns2:Request". Y "ns2:Request//ns2:AppID" busca a los descendientes "ns2:AppID" de "ns2:Request" (los hijos, los hijos de éstos, etcétera). Son efectivos cuando las composiciones utilizan recurrencia, ya que de esta forma pueden detectar si realmente la lógica es correcta. Los mutantes que están basados en esta modificación tiene por nombre el siguiente formato: m06-x-x.bpel, siempre el primer número es 06.

Pero, no es nuestro caso, porque ninguna de las composiciones utiliza recurrencia. Por lo que a la hora de estudiar los distintos mutantes que vamos a obtener de una composición, indicaremos el número de mutantes que se obtienen con esta propiedad y los excluiremos de nuestro estudio para centrarnos en otros que son realmente importantes para nuestro trabajo.

5.2. LoanApproval

A la hora de estudiar la composición LoanApproval y aplicarle los diversos operadores a su fichero BPEL original (LoanApprovalProcess.bpel), obtenemos 93 mutantes. Seguidamente, a estos mutantes les aplicamos los diversos casos de pruebas que se encuentran recogidos en el fichero "data.csv" a través del fichero BPTS basado en plantillas (LoanApprovalProcess-CSV.bpts). Una vez ejecutada esta acción, la salida que obtenemos por terminal podemos reflejarla en estos datos:

- Total: 93 mutantes
- Vivos: 11 mutantes
- Muertos: 82 mutantes
- Vivos con "//": 7 mutantes; m06-01-01.bpel, m06-02-01.bpel, m06-04-01.bpel, m06-05-01.bpel, m06-07-01.bpel, m06-10-01.bpel y m06-12-01.bpel.

En un principio, no vamos a clasificar los mutantes como equivalentes, ya que para ello se necesita estudiar más profundamente los mutantes. Por tanto, clasificaremos como equivalentes o no a alguno de los mutantes cuando les hayamos realizado las últimas pruebas con los nuevos casos de prueba que obtengamos gracias a la aplicación que desarrollaremos más adelante.

Como podemos ver, tenemos un total de 82 mutantes muertos gracias al conjunto de casos de prueba original y 11 mutantes vivos. De estos vivos, al compararlos con el fichero original, nos encontramos que siete de ellos se han generado a partir de añadir "//" en lugar de "/", por tanto, nos quedamos con los otros cuatro mutantes como objetivo de nuestro estudio.

Al comparar los mutantes con el fichero original encontramos las siguientes diferencias:

- **m04-01-01.bpel**: en la composición original nos encontramos esto como condición de un "if" llamado "If1":

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 10000)
</condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<condition>
  (number(string($processInput.input/ns0:amount)) < 10000.0)
</condition>
```

Es una diferencia que implica que entre a trabajar el asesor antes o después, cosa que puede afectar a toda la composición si trabajamos con una cantidad que esté muy cerca de ese límite.

- **m07-01-01.bpel**: en la composición original nos encontramos el siguiente fragmento de código como condición de un "if" llamado "If1":

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 10000)
</condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 10001)
</condition>
```

Se trata de una diferencia muy parecida a la anterior, sólo que en este caso se trata de un incremento en la cantidad numérica. Afectará de igual forma si estamos trabajando con una cantidad muy cerca del límite establecido.

- **m07-01-02.bpel**: en la composición original nos encontramos el siguiente fragmento de código como condición de un "if" llamado "If1":

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 10000)
</condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 9999)
</condition>
```

Se trata de una diferencia muy parecida a las anteriores, sólo que en este caso se trata de un decremento en la cantidad numérica. Afectará de igual forma si estamos trabajando con una cantidad muy cerca del límite establecido.

- **m07-01-03.bpel**: en la composición original nos encontramos el siguiente fragmento de código como condición de un "if" llamado "If1":

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 10000)
</condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 100000)
</condition>
```

En este caso, la diferencia es que la cantidad numérica tiene un 0 más, o lo que es lo mismo, se ha multiplicado por 10 la cantidad que teníamos en el caso original.

Por tanto, vamos a desarrollar relaciones metamórficas que estén centradas en las diferencias que existen entre los mutantes. Estas diferencias son: incremento de una unidad, decremento de una unidad, multiplicación por 10, cambios lógicos a la hora de tratar la cantidad establecida como límite...

5.3. MarketPlace

Vamos a hacer lo mismo con la composición MarketPlace, vamos a aplicarle los diversos operadores a su fichero BPEL original (marketplace.bpel) y obtenemos los distintos mutantes a partir de él, más concretamente, obtenemos 34 mutantes. Les aplicamos a todos los diversos casos de pruebas que se encuentran recogidos en el fichero BPTS orignal (marketplace.bpts), ya que esta composición aún no tiene BPTS basado en plantillas CSV. Y una vez ejecutada esta acción, la salida que obtenemos por terminal podemos reflejarla en estos datos:

- Total: 34 mutantes
- Vivos: 1 mutante
- Muertos: 33 mutantes
- Vivos con "//": 0 mutantes

En esta ocasión sólo tenemos que buscar la forma de matar a ese único mutante, m17-06-01.bpel. Como el resto ya han sido matados por los casos de prueba que conforman el conjunto de casos de prueba original, no tenemos que centrarnos en ellos. Por tanto, vamos a mostrar la diferencia que exite entre este mutante y el fichero BPEL original:

- **m17-06-01.bpel**: en la composición original nos encontramos esto como respuesta final:

<reply name="SellerReply" operation="submit"
partnerLink="seller" portType="tns:sellerPT"
variable="negotiationOutcome"/>

```
<reply name="BuyerReply" operation="submit"
   partnerLink="buyer" portType="tns:buyerPT"
   variable="negotiationOutcome"/>
```

Pues bien, en este mutante encontramos lo siguiente en ese mismo lugar:

<reply name="BuyerReply" operation="submit"
partnerLink="buyer" portType="tns:buyerPT"
variable="negotiationOutcome"/>

```
<reply name="SellerReply" operation="submit"
   partnerLink="seller" portType="tns:sellerPT"
   variable="negotiationOutcome"/>
```

La diferencia que hay entre ellos dos es el orden en el que se envían las respuestas. En el caso original se envía primero la respuesta del vendedor y luego la del comprado, y en el mutante, se envía primero la respuesta del comprador y luego la del vendedor.

Como podemos ver, se trata de un caso muy paticular, ya que aparentemente no debe afectar al funcionamiento de la composición. Es decir, no interviene en la lógica de los agentes, ya que sólo consiste en un cambio de orden. En este caso, como también aparece la demora de tiempo gracias al atributo “delaySequence”, vamos a desarrollar relaciones metamórficas basadas en las demoras de tiempo, y al igual que en la composición anterior, otras relaciones basadas en el incremento de una unidad, decremento de una unidad, multiplicación por 10, cambios lógicos a la hora de tratar la cantidad establecida como límite, etc.

Con la intención de poder encontrar algún caso de prueba que mate a este mutante. Si al final de su estudio, vemos que ha sido imposible matarlo, lo catalogaremos como un mutante equivalente.

5.4. MetaSearch

Por último, vamos a estudiar la composición MetaSearch, aplicando los diversos operadores a su fichero BPEL original (MetaSearch.bpel) y obteniendo los distintos mutantes a partir de él. Obtenemos un total de 706 mutantes a los cuales les aplicamos los diversos casos de pruebas que se encuentran recogidos en el fichero BPTS original (MetaSearchTest.bpts), que al igual que la composición MarketPlace, no dispone de fichero BPTS basado en plantillas CSV. Y una vez ejecutada esta acción, la salida que obtenemos por terminal podemos reflejarla en estos datos:

- Total: 706 mutantes
- Vivos: 140 mutantes
- Muertos: 566 mutantes
- Vivos con "//": 40 mutantes

En esta ocasión, tenemos una gran cantidad de mutantes vivos, aunque respecto al resto de composiciones, están en proporción al número total de mutantes. Pues bien, al igual que en LoanApproval y MarketPlace, vamos a ir estudiando las diferencias de cada mutantes respecto al fichero BPEL original. Como dijimos que los mutantes cuya variación estuviera basada en "//" no ibamos a estudiarlos y en esta ocasión son 40, vamos a estudiar finalmente un total de 100 mutantes para la composición MetaSearch.

- **m01-52-01.bpel**: en la composición original nos encontramos en el condicional llamado "GoogleHasMoreResults", que en caso de que se cumpla su condición, el origen para copiar un valor es el siguiente:

  ```
  <bpel:from>-1 + 2*$maxGoogle</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>(-1.0 + (2.0 *$div2counter_maxResults))</bpel:from>
  ```

  En este caso, cambiamos el número de resultados de Google, por el número de resultados máximos que nos ha indicado el usuario. Puede afectar en el número de vueltas que llegue a dar el bucle "forEach" que le sigue, pero, no tiene por qué, ya que se ve afectado también por otros factores.

- **m01-60-01.bpel**: en la composición original nos encontramos en el condicional llamado "GoogleHasMoreResults", que en caso de que no se cumpla su condición, el origen para copiar un valor es el siguiente:

```
<bpel:from>-1 + 2*$maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>(-1.0 + (2.0 *$div2counter_maxResults))</bpel:from>
```

Se trata de una diferencia muy parecida a la anterior, sólo que en este caso se trata de cambiar en el caso original, el número de resultados de MSN, por el número de resultados máximos que nos ha indicado el usuario. Tiene las mismas consecuencias que el mutante anterior.

- **m01-65-01.bpel**: en la composición original nos encontramos en la condición del condicional llamado "NotEnoughResultsToReachMax" la siguiente expresión:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($maxMSN + $maxMSN) < $div2counter_maxResults)
</bpel:condition>
```

En lugar de sumar el número de resultados de Google y de MSN, en el mutante estamos sumandos dos veces el número de resultados de MSN. Como podemos ver, en esta composición, el juego con los resultados va a ser uno de los puntos claves para las relaciones metamórficas.

- **m01-66-01.bpel**: en la composición original nos encontramos en la misma condición del mutante anterior, la siguiente expresión:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($added + $maxMSN) < $div2counter_maxResults)
</bpel:condition>
```

En lugar de sumar el número de resultados de Google y de MSN, en esta ocasión, en el mutante estamos sumando el número guardado en la varible "added" junto al número de resultados de MSN. La variable "added" se inicializa con el valor 0 y no se modifica antes de llegar a ese condicional, sino en el condicional llamado "Switch_11".

- **m01-67-01.bpel**: en la composición original nos encontramos en la misma condición de los casos anteriores, la siguiente expresión:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($div2counter_final + $maxMSN) < $div2counter_maxResults)
</bpel:condition>
```

En lugar de sumar el número de resultados de Google y de MSN, en esta ocasión, en el mutante estamos sumando el número guardado en la varible "div2counter_final" junto al número de resultados de MSN. La variable "div2counter_final" se inicializa en el condicional anterior a éste y se le asigna un valor u otro, según se cumpla una condición.

- **m01-69-01.bpel**: en la composición original nos encontramos en la misma condición de los casos anteriores, la siguiente expresión:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($maxGoogle + $maxGoogle) < $div2counter_maxResults)
</bpel:condition>
```

En lugar de sumar el número de resultados de Google y de MSN, en el mutante estamos sumandos dos veces el número de resultados de Google.

- **m01-70-01.bpel**: en la composición original nos encontramos en la misma condición de los casos anteriores, la siguiente expresión:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($maxGoogle + $added) < $div2counter_maxResults)
</bpel:condition>
```

En lugar de sumar el número de resultados de Google y de MSN, en esta ocasión, en el mutante estamos sumando el número guardado en la varible "added" junto al número de resultados de Google. Al igual que antes, la variable "added" se inicializa a 0 y no es modificada antes de llegar a este condicional.

- **m01-71-01.bpel**: en la composición original nos encontramos en la misma condición de los casos anteriores, la siguiente expresión:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($maxGoogle + $div2counter_final) < $div2counter_maxResults)
</bpel:condition>
```

En lugar de sumar el número de resultados de Google y de MSN, en esta ocasión, en el mutante estamos sumando el número guardado en la varible "div2counter_final" junto al número de resultados de Google. La variable "div2counter_final" se inicializa de igual forma a como hemos explicado en el mutante m01-67-01.bpel.

- **m01-77-01.bpel**: en la composición original nos encontramos en el condicional llamado "GoogleHasMoreResults", en el caso de que se cumpla el condicional, el origen del valor es el siguiente:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>($maxMSN + $maxMSN)</bpel:from>
```

En lugar de sumar el número de resultados de Google más el de MSN, suma el número de resultados de MSN dos veces.

- **m01-78-01.bpel**: en la composición original nos encontramos en la misma situación que en el caso anterior, la siguiente expresión:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>($added + $maxMSN)</bpel:from>
```

En lugar de sumar el número de resultados de Google más el de MSN, suma el valor de "added" junto al de MSN. En este caso, como el valor de "added" es 0, se le asignará siempre el valor de MSN.

- **m01-79-01.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, la siguiente expresión:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>($div2counter_final + $maxMSN)</bpel:from>
```

En lugar de sumar el número de resultados de Google más el de MSN, suma el valor guardado en "div2counter_final" junto a los resultados de MSN. En este caso, el valor de "div2counter_final" dependerá de un condicional, como ya explicamos anteriormente.

- **m01-81-01.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, la siguiente expresión:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>($maxGoogle + $maxGoogle)</bpel:from>
```

En lugar de sumar el número de resultados de Google más el de MSN, suma el número de resultados de Google dos veces. Es un caso muy parecido al del mutante m01-77-01.bpel.

- **m01-82-01.bpel**: en la composición original nos encontramos en el mismo condicional que en los casos anteriores, la siguiente expresión:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>($maxGoogle + $added)</bpel:from>
```

En lugar de sumar el número de resultados de Google más el de MSN, suma el número de resultados de Google más el valor guardado en "added". Aquí "added" sigue llegando con el valor 0, sin modificarse anteriormente.

- **m01-83-01.bpel**: en la composición original nos encontramos en el mismo condicional que en los casos anteriores, la siguiente expresión:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>($maxGoogle + $div2counter_final)</bpel:from>
```

En lugar de sumar el número de resultados de Google más el de MSN, suma el número de resultados de Google más el valor que guarda la variable "div2counter_final". Aquí el valor de "div2counter_final" dependerá de un condicional de nuevo.

- **m01-91-01.bpel**: en la composición original nos encontramos en el condicional "GoogleHasResults" que la condición es la siguiente:

```
<bpel:condition>
 ($counter <= $maxGoogle)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($counter <= $div2counter_final)
</bpel:condition>
```

El mutante se encarga de comparar si la variable "div2counter_final" es mayor o igual que la variable "counter", cuando realmente debe de compararlo con el número de resultados de Google si lo que quiere saber es si exiten resultados para mostrar.

- **m01-92-01.bpel**: en la composición original nos encontramos en la misma condición que en el mutante anterior, y nos encontramos la siguiente expresión:

```
<bpel:condition>
 ($counter <= $maxGoogle)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($counter <= $div2counter_maxResults)
</bpel:condition>
```

El mutante se encarga de comparar si la variable "div2counter_maxResults" es mayor o igual que la variable "counter", cuando realmente debe de compararlo con el número de resultados de Google si lo que quiere saber es si exiten resultados para mostrar. Es un caso muy parecido al anterior.

- **m01-95-01.bpel**: en la composición original nos encontramos en el condicional "MSN-HasResults" que la condición es la siguiente:

```
<bpel:condition>
 ($counter <= $maxMSN)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($counter <= $div2counter_final)
</bpel:condition>
```

El mutante se encarga de comparar si la variable "div2counter_final" es mayor o igual que la variable "counter", cuando realmente debe de compararlo con el número de resultados de MSN si lo que quiere saber es si exiten resultados para mostrar. Es un caso muy parecido a los anteriores, pero, trabajando con el agente MSN.

- **m01-96-01.bpel**: en la composición original nos encontramos en la misma condición que en el mutante anterior, y nos encontramos la siguiente expresión:

```
<bpel:condition>
 ($counter <= $maxMSN)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($counter <= $div2counter_maxResults)
</bpel:condition>
```

El mutante se encarga de comparar si la variable "div2counter_maxResults" es mayor o igual que la variable "counter", cuando de nuevo debe compararlo con el número de resultados de MSN.

- **m01-99-01.bpel**: en la composición original nos encontramos en el condicional "Switch_11" que la condición es la siguiente:

```
<bpel:condition>
 ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($div2counter_final = 0.0)
</bpel:condition>
```

En este caso, se comprueba el valor de la variable "div2counter_final" en lugar del valor de "added". Y en el caso del mutante, muchas de las veces no cumplirá el condicional, ya que se le asignará un valor en función de los resultados, por lo que influye en el funcionamiento de la composición.

- **m01-100-01.bpel**: en la composición original nos encontramos en la misma condición que en el mutante anterior, y nos encontramos la siguiente expresión:

```
<bpel:condition>
 ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($div2counter_maxResults = 0.0)
</bpel:condition>
```

Ocurre prácticamente lo mismo que en el mutante anterior, ya que a la hora de asignarle un valor a la variable "div2counter_maxResults", se le asigna en función de los resultados del caso de prueba, por lo que puede afectar al funcionamiento de la composición.

- **m02-02-01.bpel**: en la composición original nos encontramos en el condicional "GoogleHasMoreResults", y en el caso de que se cumpla la condición que impone, tenemos una asignación de valores. Pues bien, el origen de este valor en el caso original es el siguiente:

```
<bpel:from>-1+2 *$maxGoogle = 0)</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>-1+2 +$maxGoogle = 0)</bpel:from>
```

La única diferencia que existe entre ellos es que en lugar de hacer una multiplicación, se hace una suma y el resultado final se le asigna a la variable "div2counter_final".

- **m02-05-02.bpel**: en la composición original nos encontramos en el condicional "NotEnoughResultsToReachMax", cuya condición es la siguiente:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 $maxGoogle - $maxMSN < $div2counter_maxResults
</bpel:condition>
```

La única diferencia que existe entre ellos es que en lugar de comparar la suma entre los resultados de Google y MSN, lo que hace es restarlos, provocando probablemente un funcionamiento distinto en la composición.

- **m02-06-01.bpel**: en la composición original nos encontramos en el condicional "NotEnoughResultsToReachMax", y en caso de que se cumpla la condición establecida, se realiza una asignación cuyo origen en la composición original es el siguiente:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>$maxGoogle - $maxMSN</bpel:from>
```

La única diferencia que existe entre ellos es que asigna una resta, cuando realmente debe asignar la suma de los resultados entre Google y MSN a la variable "div2counter_maxResults".

- **m02-06-02.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior y ésta es la expresión para indicar el origen de la asignación:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>$maxGoogle * $maxMSN</bpel:from>
```

Es prácticamente igual al mutante anterior, solo que en este caso en lugar de tener el mutante una resta, tiene una multiplicación entre los resultados de Google y de MSN.

- **m02-06-03.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior y ésta es la expresión para indicar el origen de la asignación:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>$maxGoogle div $maxMSN</bpel:from>
```

En esta ocasión, se utiliza el valor que obtenemos de la división entera entre los resultados de Google y MSN para asignárselo a la variable "div2counter_maxResults".

- **m02-06-04.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior y ésta es la expresión para indicar el origen de la asignación:

```
<bpel:from>$maxGoogle + $maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>$maxGoogle mod $maxMSN</bpel:from>
```

Aquí se utiliza el resto de la división entre los resultados de Google y MSN como valor de origen para la asignación. Como todos los casos anteriores que son casi iguales a este mutante, el resultado de esta operación podrá perjudicar el funcionamiento del bucle forEach que le sigue en la composición, pero, al existir otros factores que influyen en dicho funcionamiento, puede que no sean tan importantes estos cambios.

- **m03-01-01.bpel**: en la composición original nos encontramos en el condicional "GoogleHasMoreResults", y en caso de que se cumpla la condición establecida, se realiza una asignación cuyo origen en la composición original es el siguiente:

```
<bpel:from> -1 + 2*$maxGoogle</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from> 1 + 2*$maxGoogle</bpel:from>
```

La única diferencia que existe entre ellos es que en el caso original se le resta una unidad al valor obtenido de multiplicar por dos el número de resultados de Google, y en el mutante, le sumamos una unidad. Este tipo de mutantes en los que el valor tiene un margen de diferencia tan pequeño, puede ser en ciertas ocasiones, bastante difíciles de detectar.

- **m03-02-01.bpel**: en la composición original nos encontramos en el condicional "GoogleHasMoreResults", y en caso de que no se cumpla la condición establecida, se realiza una asignación cuyo origen en la composición original es el siguiente:

```
<bpel:from> -1 + 2*$maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from> 1 + 2*$maxMSN</bpel:from>
```

Se trata prácticamente del mismo mutante que el anterior, sólo que en este caso trabajamos con el agente MSN en lugar de Google.

- **m04-03-04.bpel**, **m04-04-04.bpel**: en la composición original nos encontramos en el condicional "IsDoneAtAll" cuya condición es la siguiente:

```
<bpel:condition>
 ((boolean($done_google) = true())
  or
 (boolean($done_msn) = true()))
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ((boolean($done_google) = true())
  or
 (boolean($done_msn) >= true()))
</bpel:condition>
```

La diferencia entre ambos es que en lugar de comparar dos expresiones booleanas a través de un "=", lo hacen a través de un ">=", cosa que no debe de influir en el funcionamiento de la composición, ya que una expresión booleana va a ser siempre igual o distinta, no mayor o menor que otra. Por tanto, puede tratarse de un posible mutante equivalente.

- **m04-05-03.bpel**: en la composición original nos encontramos en el condicional "GoogleHasMoreResults" cuya condición es el siguiente:

```
<bpel:condition>$maxGoogle > $maxMSN</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>$maxGoogle >= $maxMSN</bpel:condition>
```

Con este cambio, lo único que hace es permitir que pueda ocurrir la actividad que sigue a la condición, en los casos en que el número de resultados de Google y MSN sean iguales.

- **m04-06-02.bpel**: en la composición original nos encontramos en el condicional "NotEnoughResultsToReachMax", cuya condición es la siguiente:

```
<bpel:condition>
 $maxGoogle + $maxMSN < $div2counter_maxResults
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 $maxGoogle + $maxMSN <= $div2counter_maxResults
</bpel:condition>
```

Con este cambio, lo único que hace es permitir que pueda ocurrir la actividad que sigue a la condición, en los casos en que la suma de resultados de Google más los de MSN sean iguales al valor que guarda la variable "div2counter_maxResults".

- **m04-07-03.bpel**: en la composición original nos encontramos en el condicional "IfEvenIteration", cuya condición es la siguiente:

```
<bpel:condition>
 (($div2counter mod 2) = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($div2counter mod 2) <= 0)
</bpel:condition>
```

En este mutante se da la posibilidad de que se cumpla la actividad del condicional en aquellas situaciones cuyo resto entre div2counter y 2 sea negativo. Pero esto nunca va a suceder, ya que los posibles restos de 2 es 0 y 1. Por lo que este mutante es un claro caso de equivalencia.

- **m04-10-03.bpel**: en la composición original nos encontramos en el condicional "MSNResultHasDescription", cuya condición es la siguiente:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) > 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) >= 0)
</bpel:condition>
```

En este mutante se da la posibilidad de que se cumpla la actividad del condicional en aquellas situaciones en las que el número de descripciones sea también igual a 0. Por lo que se amplía en uno la posibilidad de actuación del condicional.

- **m04-10-05.bpel**: en la composición original nos encontramos en el mismo condicional del mutante anterior, cuya condición es la siguiente:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) > 0)
</bpel:condition>
```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) != 0)
</bpel:condition>
```

  Aquí se indica que siempre que el número de descripciones sea distinto de 0, se ejecutará la actividad que sigue al condicional. Como el número de descripciones nunca va a ser inferior a 0, y este mutante indica que debe ser distinto de 0, sólo queda la posibilidad de valores que sean mayores que 0. Por lo que se trata de nuevo de un claro caso de equivalencia.

- **m04-12-04.bpel**: en la composición original nos encontramos en el condicional "Switch_7", cuya condición es la siguiente:

```
<bpel:condition>
 ($doAdd = true())
</bpel:condition>
```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($doAdd >= true())
</bpel:condition>
```

  Como ya hemos comentado anteriormente, nos encontramos de nuevo con un mutante cuya variación consiste en comparar dos valores booleanos como si fueran valores aritméticos. Es decir, comparando (además de si son distintos) si son mayores o menores. Como dos valores booleanos sólo pueden ser iguales o distintos, la aparición del ">" no afecta al funcionamiente de la composición, por lo que se trata de otro mutante equivalente.

- **m04-13-01.bpel**: en la composición original nos encontramos en el condicional "Switch_11", cuya condición es la siguiente:

```
<bpel:condition>
 ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($added < 0.0)
</bpel:condition>
```

Este cambio si puede afectar a la composición en ciertas ocasiones, ya que el valor más pequeño que puede llegar a tener "added" es 0, por lo que la condición que se impone en el mutante nunca llegaría a ejecutarse. Sólo entraría siempre en la actividad del "else".

- **m04-13-01.bpel**: en la composición original nos encontramos en la misma situación del mutante anterior, cuya condición es la siguiente:

```
<bpel:condition>
 ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($added <= 0.0)
</bpel:condition>
```

Este cambio, al contrario que el anterior, no afecta al funcionamiento de la composición. Esto es debido a que el igual permanece en los dos casos y en el mutante se añade la posibilidad de que puedan admitirse datos negativos de "added". Pero, como ya hemos comentado anteriormente, "added" no puede tomar valores menores a 0 porque lo impone la lógica de la composición, se van a ejecutar en las mismas situaciones ambas condiciones. Por tanto, se trata de un mutante equivalente.

- **m05-01-01.bpel**: en la composición original nos encontramos en el condicional "Switch_10", cuya condición es la siguiente:

```
<bpel:condition>
 (($inputVariable.payload/client:country != '')
    and
 ($inputVariable.payload/client:language != ''))
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (($inputVariable.payload/client:country != '')
   or
 ($inputVariable.payload/client:language != ''))
</bpel:condition>
```

Es un cambio bastante importante a la hora de comprobar si funciona correctamente la composición en ciertos casos de prueba, ya que en aquellos casos en los que el valor guardado en "country" o en "language" sea vacío, obtendremos una salida distinta al programa original.

- **m07-06-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_5", dentro de la asignación "CounterToOne". Pues bien, una de las copias de valores tiene como origen la siguiente expresión:

```
<bpel:from>number( 0)</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>number( 1)</bpel:from>
```

Lo único que se modifica es el valor con el que inicializamos la variable "noResult" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a 1. Realmente, lo que se está haciendo es sumar una unidad al valor inicial de la variable.

- **m07-06-02.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior y nos encontramos la siguiente expresión:

```
<bpel:from>number( 0)</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>number( -1)</bpel:from>
```

Lo único que se modifica es el valor con el que inicializamos la variable "noResult" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a -1. En este caso, lo que se está haciendo es restar una unidad al valor inicial de la variable.

- **m07-06-03.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior y nos encontramos la siguiente expresión:

  ```
  <bpel:from>number( 0)</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>number( 10)</bpel:from>
  ```

  Lo único que se modifica es el valor con el que inicializamos la variable "noResult" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a 10. Realmente, lo que se está haciendo es sumar diez unidades al valor inicial de la variable.

- **m07-06-04.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior y nos encontramos la siguiente expresión:

  ```
  <bpel:from>number( 0)</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>number( -10)</bpel:from>
  ```

  Lo único que se modifica es el valor con el que inicializamos la variable "noResult" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a -10. Realmente, lo que se está haciendo es restar diez unidades al valor inicial de la variable.

- **m07-09-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_11", dentro de la asignación "MaxNo0". Pues bien, una de las copias de valores tiene como origen la siguiente expresión:

  ```
  <bpel:from>number( 0)</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>number( 1)</bpel:from>
  ```

  Lo único que se modifica es el valor con el que inicializamos la variable "maxMSN" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a 1. Realmente, lo que se está haciendo es sumar una unidad al valor inicial de la variable.

- **m07-09-03.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior:

  ```
  <bpel:from>number( 0)</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>number( 10)</bpel:from>
  ```

  Lo único que se modifica es el valor con el que inicializamos la variable "maxMSN" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a 10. Realmente, lo que se está haciendo es sumar diez unidades al valor inicial de la variable.

- **m07-10-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_10", dentro de la asignación "MaxNo0". Pues bien, una de las copias de valores tiene como origen la siguiente expresión:

  ```
  <bpel:from>number( 0)</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>number( 1)</bpel:from>
  ```

  Lo único que se modifica es el valor con el que inicializamos la variable "maxGoogle" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a 1. Realmente, lo que se está haciendo es sumar una unidad al valor inicial de la variable.

- **m07-10-03.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior:

  ```
  <bpel:from>number( 0)</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>number( 10)</bpel:from>
  ```

  Lo único que se modifica es el valor con el que inicializamos la variable "maxGoogle" del agente cliente, ya que en lugar de inicializarse a 0, se inicializa a 10. Realmente, lo que se está haciendo es sumar diez unidades al valor inicial de la variable.

- **m07-11-01.bpel**: en la composición original nos encontramos en el condicional llamado "GoogleHasMoreResults", que en caso de que se cumpla su condición, el origen para copiar un valor es el siguiente:

  ```
  <bpel:from>- 1 + 2*$maxGoogle</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>(- 2.0 + (2.0 *$maxGoogle))</bpel:from>
  ```

  En lugar de restarle una unidad al doble de resultados de Google, se le restan dos. Es otra modificación muy parecida al mutante m01-52-01.bpel.

- **m07-11-02.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior, y nos encontramos con la siguiente expresión:

  ```
  <bpel:from>- 1 + 2*$maxGoogle</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>(- 0 + (2.0 *$maxGoogle))</bpel:from>
  ```

  En esta ocasión, la modificación que se hace es no restarle ni sumarle nada, sino dejar como valor a copiar el doble de resultados de Google.

- **m07-11-04.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, y nos encontramos con la siguiente expresión:

  ```
  <bpel:from>- 1 + 2*$maxGoogle</bpel:from>
  ```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

  ```
  <bpel:from>(- -1 + (2.0 *$maxGoogle))</bpel:from>
  ```

  En esta ocasión, la modificación que se hace es restarle un -1, o lo que es lo mismo, pasa de restarle una unidad a sumársela.

- **m07-12-01.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, y nos encontramos con la siguiente expresión:

```
<bpel:from>- 1 + 2*$maxGoogle</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>(-1 + ( 3 *$maxGoogle))</bpel:from>
```

En esta ocasión, la modificación que se hace es multiplicar el número de resultados de Google por tres, en lugar de por dos. Lo que se hace realmente es sumarle una unidad a ese valor.

- **m07-12-03.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, y nos encontramos con la siguiente expresión:

```
<bpel:from>- 1 + 2*$maxGoogle</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>(-1 + ( 22 *$maxGoogle))</bpel:from>
```

En esta ocasión, la modificación que se hace es multiplicar el número de resultados de Google por 22, en lugar de por dos. Lo que se hace realmente es añadir una cifra más a ese valor, o lo que es lo mismo, multiplicar por 11 ese valor.

- **m07-13-02.bpel**: en la composición original nos encontramos en el condicional llamado "GoogleHasMoreResults", que en caso de que no se cumpla su condición, el origen para copiar un valor es el siguiente:

```
<bpel:from>- 1 + 2*$maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>(- 0 + (2.0 *$maxMSN))</bpel:from>
```

En lugar de restarle una unidad al doble de resultados de MSN, no se le resta nada y se asigna sólo el doble de resultados de MSN. Es un mutante muy parecido a los anteriores, sólo que en este caso trabaja con el agente MSN.

- **m07-11-04.bpel**: en la composición original nos encontramos en la misma situación que en el mutante anterior, y nos encontramos con la siguiente expresión:

```
<bpel:from>- 1 + 2*$maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>(- -1 + (2.0 *$maxMSN))</bpel:from>
```

En esta ocasión, la modificación que se hace es restarle un -1, o lo que es lo mismo, pasa de restarle una unidad a sumársela al resultado de multiplicar por dos el número de resultados de MSN.

- **m07-14-01.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, y nos encontramos con la siguiente expresión:

```
<bpel:from>- 1 + 2*$maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>(-1 + ( 3 *$maxMSN))</bpel:from>
```

En esta ocasión, la modificación que se hace es multiplicar el número de resultados de MSN por tres, en lugar de multiplicarlos por 2. Lo que se hace en esta ocasión es sumarle uno a ese valor.

- **m07-14-03.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, y nos encontramos con la siguiente expresión:

```
<bpel:from>- 1 + 2*$maxMSN</bpel:from>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:from>(-1 + ( 22 *$maxMSN))</bpel:from>
```

En esta ocasión, la modificación que se hace es multiplicar el número de resultados de MSN por 22, en lugar de por dos. Lo que se hace realmente es añadir una cifra más a ese valor, o lo que es lo mismo, multiplicar por 11 ese valor.

- **m07-19-02.bpel**: en la composición original nos encontramos en el condicional llamado "MSNResultsHasDescription", cuya condición tiene la siguiente expresión:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) > 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) > -1)
</bpel:condition>
```

Se comprueba si el número de descripciones en los resultados de MSN son mayores que cero (en el caso original) o mayores que -1 (en el mutante). La diferencia es evidente, ya que el original se ejecutará siempre, excepto cuando no haya descripciones; pero en cambio, el mutante se ejecutará siempre, sin excepciones. Porque el mínimo de descripciones que puede haber en un caso de prueba es cero.

- **m07-19-04.bpel**: en la composición original nos encontramos en la misma situación que en el caso anterior, cuya expresión es la siguiente:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) > 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) > -10)
</bpel:condition>
```

Ocurre lo mismo que en el mutante anterior, porque al cambiar 0 por -10, hace que se ejecute en todos los casos posibles por la misma explicación que hemos dado anteriormente.

- **m07-20-02.bpel**: en la composición original nos encontramos en la misma situación que en el caso anterior, cuya expresión es la siguiente:

```
<bpel:condition>
 ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($added = -1)
</bpel:condition>
```

En este caso, se comprueba si el valor de la variable “added” es igual a -1, en el mutante. Esto implica que este condicional se ejecute siempre ya que el valor más pequeño que va a tomar “added” va a ser 0.

- **m07-20-03.bpel**: en la composición original nos encontramos en la misma situación que en el caso anterior, cuya expresión es la siguiente:

```
<bpel:condition>
 ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($added = 10)
</bpel:condition>
```

En este caso, se comprueba si el valor de la variable “added” es igual a 10, en el mutante. Esto implica que este condicional se ejecute siempre y cuando “added” llegue al valor 10, debido a que haya suficientes resultados para mostrar.

- **m07-20-03.bpel**: en la composición original nos encontramos en la misma situación que en los casos anteriores, cuya expresión es la siguiente:

```
<bpel:condition>
 ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
 ($added = -10)
</bpel:condition>
```

En este caso, se comprueba si el valor de la variable “added” es igual a -10, en el mutante. Esto implica que este condicional se ejecute siempre ya que el valor más pequeño que va a tomar “added” va a ser 0.

- **m11-02-01.bpel**: en la composición original nos encontramos en el bucle "forEach" que se llama "ScanResultsForMatch", cuya expresión para indicar esta actividad es la siguiente:

```
< bpel:forEach counterName="currentResultNumber"
  name="ScanResultsForMatch"  parallel="no">
     ...
</bpel:forEach>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<forEach  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  counterName="currentResultNumber"
  name="ScanResultsForMatch"
   parallel="yes">
     ...
</forEach>
```

Aparentemente parece que se realizan dos cambios, pero, realmente es sólo uno. El primero que es la eliminación de "bpel:" indicando el espacio de nombres, es equivalente a la cabecera que se incluye como xmlns. Y luego, el verdadero cambio es poner la propiedad parallel a "yes", cuando en el mutante original está a "no". Como este foreach sólo contiene una actividad que es un condicional, el cual si se cumple se realiza una asignación y en caso contrario no se realiza nada, puede tratarse de un mutante equivalente porque se realice o no en paralelo, siempre se va a ejecutar sólo una actividad.

- **m12-01-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_15", cuya expresión para indicar esta actividad es la siguiente:

```
<bpel: sequence name="Sequence_15">...</bpel: sequence>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel: flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
  name="Sequence_15">
     ...
</bpel: flow>
```

Ocurre algo parecido al caso anterior, parece que se realizan dos cambios, pero realmente es sólo uno. Se cambia la actividad "sequence" por "flow" (a la cual se le añade las cabeceras correspondientes). Pero, si miramos el conjunto de actividades que recoge esta actividad, se trata de dos copias de datos independientes entre ellas. Por lo que probablemente se trate también de un mutante equivalente, debido a que se ejecuten secuencial o concurrentemente, el resultado que obtendremos es el mismo: dos copias de datos.

- **m12-03-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_3", cuya expresión para indicar esta actividad es la siguiente:

```
<bpel: sequence name="Sequence_3">...</bpel: sequence>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel: flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
  name="Sequence_3">
    ...
</bpel: flow>
```

Es algo parecido al caso anterior, sólo que cambiando de secuencia. En este caso, el conjunto de actividades que recoge son una serie de asignaciones independientes entre sí, la invocación de una actividad y luego otra asignación en la cual se refleja que se ha completado la secuencia completa sin problemas. De nuevo puede tratarse de un posible mutante equivalente, ya que si se cambia la ejecución a forma concurrente, se asignarán todos los valores y se invocará la actividad. Y en el caso de que se produjera algún error, se detendría la ejecución y sería capturado por el manejador de errores; por tanto, el resultado final debe ser el mismo.

- **m12-07-01.bpel**: en la composición original nos encontramos en la secuencia "LoadGoogleResult", cuya expresión para indicar esta actividad es la siguiente:

```
<bpel: sequence name="LoadGoogleResult">...</bpel: sequence>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel: flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
```

```
    name="LoadGoogleResult">
      ...
</bpel:flow>
```

De nuevo la misma situación, nos encontramos con una secuencia que recoge una serie de asignaciones de valores independientes entre sí a la hora de recoger los distintos resultados de Google. Si éstas no se afectan entre ellas al cambiar de valor, el resultado debe ser el mismo si se ejecuta de forma secuencial o de forma concurrente. Por lo que puede tratarse de otro posible mutante equivalente.

- **m12-08-01.bpel**: en la composición original nos encontramos en la secuencia "LoadMSNResult", cuya expresión para indicar esta actividad es la siguiente:

```
<bpel:sequence name="LoadMSNResult">...</bpel:sequence>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
   name="LoadMSNResult">
      ...
</bpel:flow>
```

Es lo mismo que en el caso anterior, pero, en esta ocasión trabajando con los resultados del agente MSN. Otro posible mutante equivalente.

- **m14-01-01.bpel**: en la composición original nos encontramos en el condicional "Switch_10" lo siguiente:

```
<bpel:if name="Switch_10">
      ...
 </bpel:assign>
  <bpel:else>
<bpel:assign name="UseDefaulten-US">
 <bpel:copy>
  <bpel:from>'en-US'</bpel:from>
  <bpel:to part="parameters"
    variable="MSNSearch_Search_InputVariable">
   <bpel:query>ns2:Request/ns2:CultureInfo</bpel:query>
  </bpel:to>
```

```
 </bpel:copy>
</bpel:assign>
</bpel:else>
</bpel:if>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:if name="Switch_10">
    ...
 </bpel:assign>
</bpel:if>
```

El operador se encarga de eliminar la parte del "else" en el mutante, cosa que afectará notablemente al resultado de la ejecución de la composición, ya que los casos que no cumplan la condición tendrán la variable "CultureInfo" vacía en el agente MSN, cuando no debe ser así.

- **m14-07-01.bpel**: en la composición original nos encontramos en el condicional "MSNResultHasDescription" lo siguiente:

```
<bpel:if name="MSNResultHasDescription">
    ...
   <bpel:else>
 <bpel:empty/>
</bpel:else>
 </bpel:if>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:if name="MSNResultHasDescription">
     ...
</bpel:if>
```

En este caso, de nuevo el mutante aparece sin la parte del else de un condicional en particular. Esta vez, el else no recogía ninguna serie de actividades ya que estaba declarado como "empty", por lo que si no aparece en el mutante, debemos obtener el mismo resultado que en el original. Así que debe tratarse de un mutante equivalente.

- **m14-10-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_16" lo siguiente:

```
<bpel:if name="Sequence_16">
    ...
   <bpel:else>
 <bpel:empty/>
</bpel:else>
 </bpel:if>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:if name="Sequence_16">
    ...
</bpel:if>
```

Ocurre lo mismo que en el caso anterior, el else no aparece en el mutante, pero, no contenía ninguna actividad. Por lo que nos encontramos de nuevo ante otro posible mutante equivalente.

- **m17-01-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_15" lo siguiente:

```
<bpel:sequence name="Sequence_15">
  <bpel:assign name="CreateFault">
    ...
  </bpel:assign>
   <bpel:reply faultName="client:MetaSearchFault" name="ReplyFault"
 operation="process" partnerLink="client"
 portType="client:MetaSearch" variable="generalFault"/>
</bpel:sequence>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:sequence name="Sequence_15">
    <bpel:reply faultName="client:MetaSearchFault" name="ReplyFault"
  operation="process" partnerLink="client"
  portType="client:MetaSearch" variable="generalFault"/>
  <bpel:assign name="CreateFault">
    ...
  </bpel:assign>
</bpel:sequence>
```

La diferencia existente entre ellos, es que se envía la respuesta al principio de la secuencia en el caso del mutante, y en el caso original se envía al final.

- **m17-06-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_3" lo siguiente:

```
<bpel:sequence name="Sequence_3">
    ...
    <bpel:invoke inputVariable="Google_doGoogleSearch_InputVariable"
  name="Google" operation="doGoogleSearch"
  outputVariable="Google_doGoogleSearch_OutputVariable"
  partnerLink="Google" portType="ns1:GoogleSearchPort"/>
    ...
</bpel:sequence>
```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:sequence name="Sequence_3">
    ...
    <bpel:invoke inputVariable="Google_doGoogleSearch_InputVariable"
  name="Google" operation="doGoogleSearch"
  outputVariable="Google_doGoogleSearch_OutputVariable"
  partnerLink="Google" portType="ns1:GoogleSearchPort"/>
</bpel:sequence>
```

  La diferencia existente entre ellos, es que en el caso original se invoca al agente Google y una vez recibida su respuesta, ponemos la variable "done_Google" a true. En el caso del mutante, ponemos esta variable antes a true y ya luego, invocamos al agente Google.

- **m17-12-01.bpel**: en la composición original nos encontramos en la secuencia "Sequence_4" lo siguiente:

```
<bpel:sequence name="Sequence_4">
    ...
    <bpel:invoke inputVariable="MSNSearch_Search_InputVariable"
  name="MSNSearch" operation="Search"
  outputVariable="MSNSearch_Search_OutputVariable"
  partnerLink="MSN" portType="ns2:MSNSearchPortType"/>
    ...
</bpel:sequence>
```

  Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:sequence name="Sequence_4">
    ...
```

```
    <bpel:invoke inputVariable="MSNSearch_Search_InputVariable"
    name="MSNSearch" operation="Search"
    outputVariable="MSNSearch_Search_OutputVariable"
    partnerLink="MSN" portType="ns2:MSNSearchPortType"/>
</bpel:sequence>
```

Ocurre lo mismo que en el mutante anterior, pero, con el agente MSN.

- **m17-23-01.bpel**: en la composición original nos encontramos en la parte en la que se encuentran los condicionales "GoogleHasMoreResults" y "NotEnoughResultsToReachMax", y en el caso original aparece esto:

```
...
<bpel:if name="GoogleHasMoreResults">
...
<bpel:if name="NotEnoughResultsToReachMax">
...
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
...
<bpel:if name="NotEnoughResultsToReachMax">
...
<bpel:if name="GoogleHasMoreResults">
...
```

La diferencia entre el caso original y el mutante es que el orden de los condicionales cambia. Es algo que puede afectar a la hora de ejecutarse los bucles forEach del condicional "NotEnoughResultsToReachMax", ya que dependen de la variable "div2counter_final" que toma valores en el condicional "GoogleHasMoreResults".

- **m20-01-01.bpel**: en la composición original nos encontramos en el manejador de errores global de la composición:

```
...
</bpel:variables>
<bpel:faultHandlers>
<bpel:catchAll>
<bpel:sequence name="Sequence_15">
...
</bpel:faultHandlers>
<bpel:sequence>
...
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 ...
</bpel:variables>

<bpel:sequence>
...
```

En el mutante no aparece ese manejador de errores, por lo que a la hora de tratar un caso de prueba con errores, el resultado de la ejecución de ambos será distinto.

- **m26-01-01.bpel**: de nuevo tratamos la parte en la que se encuentra el manejador de errores global, y tenemos lo siguiente:

```
<bpel:faultHandlers>
  <bpel:catchAll>
     <bpel:sequence name="Sequence_15">
 ...
</bpel:faultHandlers>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:faultHandlers>
  <bpel:catchAll>
     <bpel:empty xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:uca="http://www.uca.es/xpath/2007/11"
  xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"/>
</bpel:faultHandlers>
```

Es otro caso de modificación en el manejador de errores global, y en éste concretamente se sustituye todo su contenido, por la actividad “empty”. Lo que provocará que al trabajar con un caso de prueba erróneo, no obtengamos un resultado correcto.

- **m26-02-01.bpel**: nos encontramos en la misma situación que en los casos anteriores, y tenemos lo siguiente:

```
<bpel:faultHandlers>
  <bpel:catchAll>
    <bpel:sequence name="Sequence_15">
       <bpel:assign name="CreateFault">
     ...
```

```
    </bpel:assign>
        <bpel:reply faultName="client:MetaSearchFault"
          name="ReplyFault" operation="process"
          partnerLink="client" portType="client:MetaSearch"
          variable="generalFault"/>
      </bpel:sequence>
  </bpel:faultHandlers>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 <bpel:faultHandlers>
  <bpel:catchAll>
    <bpel:sequence name="Sequence_15">
      <bpel:reply faultName="client:MetaSearchFault"
        name="ReplyFault" operation="process"
        partnerLink="client" portType="client:MetaSearch"
        variable="generalFault"/>
    </bpel:sequence>
  </bpel:faultHandlers>
```

En este caso no se realiza ninguna acción dentro del manejador, excepto el envío de la respuesta. Al igual que en los casos anteriores, al trabajar de forma distinta, obtendremos una salida que no es la correcta.

- **m26-03-01.bpel**: nos encontramos en la misma situación que en los casos anteriores, y tenemos lo siguiente:

```
<bpel:faultHandlers>
  <bpel:catchAll>
    <bpel:sequence name="Sequence_15">
      <bpel:assign name="CreateFault">
      ...
      </bpel:assign>
       <bpel:reply faultName="client:MetaSearchFault"
  name="ReplyFault" operation="process"
  partnerLink="client" portType="client:MetaSearch"
  variable="generalFault"/>
    </bpel:sequence>
  </bpel:faultHandlers>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:faultHandlers>
 <bpel:catchAll>
   <bpel:sequence name="Sequence_15">
     <bpel:assign name="CreateFault">
     ...
     </bpel:assign>
   </bpel:sequence>
</bpel:faultHandlers>
```

Es una situación parecida a la anterior, sólo que en ésta sí se realizan las asignaciones correspondientes para recoger la información del error, pero, no llega a mandarse la respuesta. Por lo que de nuevo, obtendremos una salida errónea.

- **m26-19-01.bpel**: en la composición original nos encontramos en el condiciona "Switch_10" en el cual nos encontramos en la parte del "else" lo siguiente:

```
<bpel:else>
   <bpel:assign name="UseDefaulten-US">
 ...
</bpel:assign>
 </bpel:else>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 <bpel:else>
   <bpel:empty xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"/>
</bpel:else>
```

Esta parte se encarga de asignarle a la variable "cultureInfo" de MSN un valor determinado. Si en lugar de las asignaciones, añadimos una actividad "empty", es decir, que no haga nada, obtendremos una salida errónea.

- **m31-01-01.bpel**: nos encontramos de nuevo en la parte inicial de la composición donde se encuentra el manejador de errores, y nos encontramos lo siguiente:

```
<bpel:faultHandlers>
  <bpel:catchAll>
     <bpel:sequence name="Sequence_15">
 ...
```

```
</bpel:sequence>
</bpel:faultHandlers>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 <bpel:faultHandlers>
  <bpel:catchAll>
     <bpel:exit xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:uca="http://www.uca.es/xpath/2007/11"
    xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"/>
</bpel:faultHandlers>
```

La diferencia que existe entre ambos, es que en caso de que se esté trabajando con un caso erróneo y lo capture el manejador de la composición original, se ejecutarán una serie de asignaciones y enviará un mensaje con el error que se ha producido. Y si pasara lo mismo en el caso del mutante, al capturarlo el manejador, simplemente se saldría de la ejecución acabándola, sin llegar a realizar otra actividad.

- **m31-02-01.bpel**: nos encontramos de nuevo en la parte inicial de la composición donde se encuentra el manejador de errores, y nos encontramos lo siguiente:

```
<bpel:faultHandlers>
  <bpel:catchAll>
    <bpel:sequence name="Sequence_15">
       <bpel:assign name="CreateFault">
 ...
 </bpel:assign>
      <bpel:reply faultName="client:MetaSearchFault"
        name="ReplyFault" operation="process"
        partnerLink="client" portType="client:MetaSearch"
        variable="generalFault"/>
    </bpel:sequence>
</bpel:faultHandlers>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 <bpel:faultHandlers>
  <bpel:catchAll>
    <bpel:sequence name="Sequence_15">
       <bpel:exit xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:uca="http://www.uca.es/xpath/2007/11"
    xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"/>
      <bpel:reply faultName="client:MetaSearchFault"
        name="ReplyFault" operation="process"
        partnerLink="client" portType="client:MetaSearch"
        variable="generalFault"/>
    </bpel:sequence>
</bpel:faultHandlers>
```

En esta ocasión, en lugar se ejecutarse las actividades de asignación, el manejador pone en ejecución una actividad "exit", parando todo lo que esté ejecutandose en ese momento y produciendo una salida errónea respecto a la original.

- **m31-03-01.bpel**: nos encontramos de nuevo en la parte inicial de la composición donde se encuentra el manejador de errores, y nos encontramos lo siguiente:

```
<bpel:faultHandlers>
  <bpel:catchAll>
    <bpel:sequence name="Sequence_15">
      <bpel:assign name="CreateFault">
      ...
      </bpel:assign>
      <bpel:reply faultName="client:MetaSearchFault"
  name="ReplyFault" operation="process"
  partnerLink="client" portType="client:MetaSearch"
  variable="generalFault"/>
    </bpel:sequence>
</bpel:faultHandlers>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 <bpel:faultHandlers>
  <bpel:catchAll>
    <bpel:sequence name="Sequence_15">
      <bpel:assign name="CreateFault">
      ...
      </bpel:assign>
      <bpel:exit xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:uca="http://www.uca.es/xpath/2007/11"
    xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"/>
    </bpel:sequence>
</bpel:faultHandlers>
```

Aquí ejecuta las asignaciones de todas las variables, guardando el correspondiente mensaje de error, pero, en ligar de enviar la respuesta pone de nuevo en ejecución una actividad "exit", parando todo lo que se esté ejecutando en ese momento y produciendo una salida errónea respecto a la original.

- **m31-19-01.bpel**: en la composición original nos encontramos en el condiciona "Switch_10" en el cual nos encontramos en la parte del "else" lo siguiente:

```
<bpel:else>
   <bpel:assign name="UseDefaulten-US">
 ...
</bpel:assign>
 </bpel:else>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 <bpel:else>
   <bpel:exit xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:uca="http://www.uca.es/xpath/2007/11"
    xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"/>
</bpel:else>
```

En este mutante se sustituyen todas las actividades que se encuentran recogidas dentro del "else" por una actividad "exit". De forma que una vez que se ejecute un caso que no cumple la condición impuesta por el condicional, se pararán todas las actividades que se estén ejecutando y la salida que obtendremos no se corresponderá con la original.

- **m32-01-01.bpel, m34-01-01.bpel**: en la composición original nos encontramos en el mismo condicional del mutante anterior, cuya condición es la siguiente:

```
<bpel:condition>
     (($inputVariable.payload/client:country != '')
 and
($inputVariable.payload/client:language != ''))
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
 <bpel:condition>
    true()
</bpel:condition>
```

En esta ocasión, el mutante sustituye la condición de que el atributo “country” y “language” sean distintos de vacío, por “true”. Es decir, que la condición siempre se va a cumplir y en los casos en que uno de esos atributos esté vacío, no va a funcionar tal y como debe.

- **m32-08-01.bpel, m33-10-01.bpel, m34-12-01.bpel**: estos tres mutantes son iguales, y la diferencia respecto al original se encuentra en la condición del condicional llamado “GoogleHasResults” que es la siguiente expresión:

```
<bpel:condition>
    ($counter <= $maxGoogle)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
    true()
</bpel:condition>
```

De nuevo, el mutante sustituye la condición de que el valor de “counter” sea menor o igual que “maxGoogle”, por “true”. Es decir, que la condición siempre se va a cumplir y en los casos en que uno de esos atributos esté vacío, no va a funcionar tal y como debe.

- **m32-09-01.bpel, m33-11-01.bpel, m34-13-01.bpel**: estos tres mutantes son iguales, y la diferencia respecto al original se encuentra en la condición del condicional llamado “MSNHasResults” que es la siguiente expresión:

```
<bpel:condition>
    ($counter <= $maxMSN)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
    true()
</bpel:condition>
```

Es la misma diferencia que en el caso anterior, solo que en esta ocasión se trata del agente MSN en lugar del agente Google.

- **m32-10-01.bpel, m33-12-01.bpel, m34-14-01.bpel**: estos tres mutantes son iguales, y la diferencia respecto al original se encuentra en la condición del condicional llamado “MSNResultHasDescription” que es la siguiente expresión:

```
<bpel:condition>
    (count($tempMSNResult/ns2:Description) > 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
    true()
</bpel:condition>
```

La condición lo que hace es comprobar si el número de descripciones de los resultados de MSN es mayor que 0. Si en lugar de eso, ponemos la expresión "true", el condicional se cumplirá siempre, cosa que no debe si se está trabajando con un caso de prueba cuyos resultados de MSN no contengan descripción.

- **m32-13-02.bpel, m33-15-02.bpel, m34-17-02.bpel**: estos tres mutantes son iguales, y la diferencia respecto al original se encuentra en la condición del condicional llamado "Switch_11" que es la siguiente expresión:

```
<bpel:condition>
    ($added = 0)
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
    false()
</bpel:condition>
```

En el caso original se comprueba si el valor de "added" es 0. En ese caso se copia en la variable "result" del cliente (que realmente es el primer resultado de la variable "result"), cosa que siempre ocurre en la primera iteración, ya que "added" se inicializa a 0. Y si no se cumple, se copia en el resultado [added+1] de la variable "result". Pero en el mutante, se pone como condición "false", es decir, que nunca se va a cumplir, así que siempre se va a copiar en el resultado [added+1] independientemente de su valor. De forma que vamos a obtener la misma salida que la composición original, lo que indica que se trata de mutantes equivalentes.

- **m33-01-01.bpel, m34-02-01.bpel**: en la composición original nos encontramos en el condicional "Switch_10", cuya condición es la siguiente:

```
<bpel:condition>
    (($inputVariable.payload/client:country != '')
      and
    ($inputVariable.payload/client:language != ''))
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
    (true()
      and
    ($inputVariable.payload/client:language != ''))
</bpel:condition>
```

En esta ocasión, el mutante sustituye parte de la condición, indicando en este caso que la variable "country" del cliente siempre va a ser distinta de vacío, a través de la expresión "true". Cosa que no es cierta porque pueden darse casos en los que sí se encuentre vacío y entonces no obtengamos la misma salida que la composición original.

- **m33-02-01.bpel, m34-03-01.bpel**: nos encontramos en la misma situación que en el caso anterior, en el cual tenemos la siguiente expresión:

```
<bpel:condition>
    (($inputVariable.payload/client:country != '')
      and
    ($inputVariable.payload/client:language != ''))
</bpel:condition>
```

Y en este mutante que estamos estudiando encontramos lo siguiente en ese mismo lugar:

```
<bpel:condition>
    (($inputVariable.payload/client:country != '')
      and
    true())
</bpel:condition>
```

Es la misma situación anterior, sólo que en este caso el mutante indica que siempre va a ser distinto de vacío la variable "language". Por tanto, puede ocurrir lo mismo que ya hemos comentado anteriormente: el caso que tenga la variable "language" vacía no obtendrá los mismos resultados que con el programa original.

6 Generación de BPTS basado en plantillas con ficheros CSV

Otro de los objetivos que queremos conseguir en este trabajo es la generación de los ficheros BPTS basado en plantillas de las composiciones que aún no lo poseen. Estas composiciones son MarketPlace y MetaSearch. Para poder realizarlas, he utilizado el BPTS basado en plantillas de la composición LoanApproval como orientación.

Uno de los aspectos necesarios de explicar de la composición LoanApproval es el uso del valor **"silent"**. En los casos de pruebas que queremos reflejar que uno de los agentes no entra en acción por algún motivo, asignamos el valor "silent" a esa variable. De esta forma, indicamos que ese agente no actúa en ese caso de prueba y no dejamos esa posición vacía dentro del fichero CSV.

A continuación, vamos a explicar cómo hemos desarrollado el fichero de cada composición, los problemas que nos hemos ido encontrando a la hora de implementarlos y las soluciones que hemos aportado para conseguir resolverlos.

6.1. MarketPlace

La composición de MarketPlace tiene una serie de casos de prueba bastante sencillos, siempre tienen la misma estructura y sólo se juega con el valor que pueden llegar a tener las variables. Esto nos ayudó a la hora de implementar el fichero, ya que la estructura al ser siempre la misma, no íbamos a tener ningún problema para poder aplicar a la plantilla todos los casos de prueba que quisiéramos.

Estudiando el fichero BPTS original, llegamos a la conclusión de que había un total de ocho variables que eran importantes para cada caso de prueba. A estas variables les hemos dado los siguientes nombres:

- Sel_Item: se corresponde con el producto que ofrece el vendedor al cliente.
- Sel_Price: representa el precio de venta que pone el vendedor al producto que hemos indicado anteriormente.
- Sel_Value: es el resultado final del negocio que se ha establecido entre el comprador y el vendedor, debe coincidir con el del comprador.

- Sel_delay: es la demora de tiempo que tiene el agente que representa al vendedor.
- Buy_Item: se corresponde con el producto que quiere comprar el cliente.
- Buy_Price: representa el precio que ofrece el comprador por el producto que ha indicado anteriormente a través de “Buy_Item”.
- Buy_Value: es el resultado final del negocio que se ha establecido entre el comprador y el vendedor, debe coincidir con el del vendedor.
- Buy_delay: es la demora de tiempo que tiene el agente que representa al comprador.

Uno de los problemas que nos encontramos a la hora de implementar el fichero BPTS fue la forma de reflejar la demora de tiempo de cada uno de los agentes. En los casos de prueba originales se utilizaba el atributo “delaySequence” para indicar las distintas demoras que iban a tener los agentes, de forma que si se quería que el mismo caso de prueba se ejecutase dos veces con demoras distintas, todo se reflejaba en esa actividad con los tiempos separados por comas. Como ejemplo mostramos el siguiente fragmento de código:

```
<tes:clientTrack>
  <tes:sendReceive service="tns:marketplaceSeller"
     port="seller" operation="submit">
      <tes:send fault="false"  delaySequence="0,2">
      ...
</tes:clientTrack>
```

Con esto se permitía que se ejecutara una primera vez con 0 unidades de tiempo de demora; y una segunda vez, con 2 segundos de demora. Al intentar trasladar este atributo a la plantilla, fue imposible trabajar con él. Al ver que no encontraba ninguna solución, expuse el problema en uno de los seminarios del grupo de investigación (30 de Mayo de 2011) y Antonio Domínguez me indicó el por qué no encontraba una solución.

La razón era porque BPELUnit no estaba preparado para poder trabajar con retrasos y condiciones XPath por plantillas. La solución para poder adaptarlas era modificar BPELUnit de forma que no pusiera ningún problema y Antonio se encargó de buscar una solución a esta adaptación. El día 3 de Agosto de 2001, nos informó de que finalmente había conseguido incluir esta propiedad en BPELUnit. En vez de extender delaySequence para admitir variables XPath, incluyó un nuevo atributo llamado “delay” que sólo podía contener una expresión XPath que genere un valor numérico.

De forma que si queremos expresar las demoras expuestas anteriormente, con este nuevo atributo en el ejemplo del fragmento anterior, nos quedaría de esta forma:

```
<tes:clientTrack>
```

```
    <tes:sendReceive service="tns:marketplaceSeller"
       port="seller" operation="submit">
        <tes:send fault="false"  delay="$Sel_Delay">
        ...
  </tes:clientTrack>
```

Sólo pordríamos guardar un valor numérico en Sel_Delay, pero, esto no impide que no se pueda representar la demora de igual forma que en el caso anterior. Tendríamos un primer caso con el valor 0 guardado en la variable correspondiente, y en el siguiente caso, el valor 2. Y de esta forma, no tenemos problemas para representar la demora de tiempo con la plantilla y el fichero data.scv.

Éste fue el único problema que me encontré a la hora de adaptar todos los casos de prueba del fichero BPTS original. De forma que la plantilla quedó finalmente de la siguiente manera (marketplace-CSV.bpts):

```
<?xml version="1.0" encoding="UTF-8"?>
<tes:testSuite
   xmlns:tns="http://docs.active-endpoints.com/activebpel/
         sample/wsdl/marketplace/2006/09/marketplace.wsdl"
   xmlns:tes="http://www.bpelunit.org/schema/testSuite">
  <tes:name>MarketPlace</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
  <tes:deployment>
    <tes:put name="marketplace" type="activebpel">
      <tes:wsdl>marketplace.wsdl</tes:wsdl>
      <tes:property name="BPRFile">marketplace.bpr</tes:property>
    </tes:put>
    <tes:partner name="Buyer" wsdl="marketplace.wsdl"/>
  </tes:deployment>
  <tes:testCases>
    <tes:testCase name="MainTemplate" basedOn="" abstract="false"
         vary="true">
      <tes:setUp>
        <tes:dataSource type="csv" src="data.csv">
          <tes:property name="separator">,</tes:property>
        </tes:dataSource>
      </tes:setUp>

      <tes:clientTrack>
        <tes:sendReceive
           service="tns:marketplaceSeller"
           port="seller" operation="submit">
          <tes:send fault="false" delay="$Sel_Delay">
```

```
            <tes:template>
              <tns:inventoryItem> $Sel_Item</tns:inventoryItem>
              <tns:askingPrice> $Sel_Price</tns:askingPrice>
            </tes:template>
          </tes:send>

          <tes:receive fault="false">
            <tes:condition>
              <tes:expression>outcome</tes:expression>
              <tes:value> $Sel_Value</tes:value>
            </tes:condition>
          </tes:receive>
        </tes:sendReceive>
      </tes:clientTrack>

      <tes:partnerTrack name="Buyer">
        <tes:sendReceive
           service="tns:marketplaceBuyer"
           port="buyer" operation="submit">
          <tes:send fault="false" delay=" $Buy_Delay">
            <tes:template>
              <tns:inventoryItem> $Buy_Item</tns:inventoryItem>
              <tns:askingPrice> $Buy_Price</tns:askingPrice>
            </tes:template>
          </tes:send>

          <tes:receive fault="false">
            <tes:condition>
              <tes:expression>outcome</tes:expression>
              <tes:value> $Buy_Value</tes:value>
            </tes:condition>
          </tes:receive>
        </tes:sendReceive>
      </tes:partnerTrack>
    </tes:testCase>
  </tes:testCases>
</tes:testSuite>
```

Y el conjunto de casos de prueba originales con los que comenzamos a trabajar son los siguientes, los cuales se encuentran recogidos en el fichero “data.csv”:

Columnas: Sel_Item, Sel_Price, Sel_Value, Sel_Delay, Buy_Item, Buy_Price, Buy_Value, Buy_Delay
Caso 1.1: Takuan, 100, Deal Successful, 2, Takuan, 150, Deal Successful, 0
Caso 1.2: Takuan, 100, Deal Successful, 0, Takuan, 150, Deal Successful, 2

Caso 2.1: Takuan, 200, Deal Failed, 0, Takuan, 120, Deal Failed, 2
Caso 2.2: Takuan, 200, Deal Failed, 2, Takuan, 120, Deal Failed, 0
Caso 3.1: Takuan, 190, Deal Successful, 0, Takuan, 190, Deal Successful, 2
Caso 3.2: Takuan, 190, Deal Successful, 2, Takuan, 190, Deal Successful, 0
Caso 4: Takuan, 190, Deal Cancelled, 0, silent, 0, silent, 0
Caso 5.1: Takuan, 190, Deal Cancelled, 10, Takuan, 120, Deal Cancelled, 0
Caso 5.2: Takuan, 190, Deal Cancelled, 10, Takuan, 120, Deal Cancelled, 0

Son cinco casos de prueba, el primer caso de prueba está dividido en dos debido a las dos demoras de tiempo que tiene. El segundo y tercer caso también están divididos en dos; el cuarto caso no tenía demora de tiempo, por lo que están los valores temporales a 0 y el quinto caso también está dividido en dos. El valor "silent" del cuarto caso tiene la misma funcionalidad que el "silent" de la composición LoanApporval, simula un valor nulo para esa variable.

6.2. MetaSearch

A la hora de desarrollar el fichero BPTS basado en plantillas de esta composición, nos encontramos con más problemas que los encontrados en la composición anterior. Además del problema del atributo "delaySequence", nos vimos en la situación de que la estructura de los casos de prueba podían variar de unos a otros.

La estructura general de un caso de prueba de esta composición es la siguiente: unos datos proporcionados por el cliente, una serie de condiciones respecto a los resultados que se quieren obtener (una con el número de resultados totales y otras con las características de estos), los resultados del buscador Google y los resultados del buscador MSN. Pues bien, en algunos de los casos de prueba originales, nos encontramos situaciones tales como que el agente Google no aparece (o el agente MSN), aparecen más condiciones o resultados respecto a otro caso de prueba... Por lo que tuvimos que pensar cómo permitir que la plantilla admitiera todas estas posibilidades.

En un primer intento de adaptación, se pensó en utilizar código Velocity, pero, al utilizar condiciones respecto al número de resultados y al valor de las variables, MuBPEL no era capaz de ejecutar correctamente el programa original frente al conjunto de casos de prueba a través de la plantilla. Por lo que, después de varios intentos y modificaciones, decidimos simplificar las pruebas con otras opciones y una vez que éstas funcionen, retomar como trabajo futuro la introducción de código Velocity.

Otro intento de adaptación, esta ocasión con las condiciones de los resultados, fue reunir en una sola condición todas las posibles condiciones adicionales que no fuesen la que indica el número de resultados totales que queremos. Es decir, reunir en una expresión XPath todas las condiciones que queríamos que se cumplieran en los resultados que deseábamos obtener. Por un lado, de nuevo tuvimos problemas a la hora de ejecutarlo con MuBPEL, ya que en ciertas ocasiones daba problemas de ejecución. Y por otro lado, los resultados que obteníamos en los

casos que no daban problemas, eran iguales que los resultados que obteníamos en esos mismos casos de prueba, pero, sin condiciones. Por lo que rechazamos también esta opción, primero para evitar un nuevo problema y luego, para simplificar la complejidad del fichero.

Por lo que finalmente, pensamos en realizar una plantilla algo más simple, pero, que nos permitiera aplicar casos de prueba que nos proporcionaran resultados relevantes. Así que decidimos implementar una plantilla que reflejara todos los datos proporcionados por el usuario, la condición que indica el número de resultados que vamos a tener finalmente, tres resultados por parte del buscador Google y tres resultados por parte del buscador MSN.

De forma que el conjunto de variables que tenemos que representar en cada caso de prueba llevan los siguientes nombres:

- Query: palabras claves de la consulta que quiere realizar el usuario.
- Languaje: idioma en el que realiza la consulta.
- Country: país desde el que realiza la consulta.
- Max_Results: número máximo de resultados que quiere el usuario.
- Count: resultados que obtenemos finalmente entre los dos buscadores. Debe ser menor o igual que el valor indicado a través de "Max_Results".
- Total: número de resultados del buscador MSN.
- Source: tipo de fuente que quiere buscar (Web, News, Spelling...).
- Offset: guarda el desplazamiento a partir del primer resultado que sirve para paginación.
- Client_Delay: guarda la demora temporal que tiene el agente del cliente.
- Google_Delay: guarda la demora temporal que tiene el agente del buscador Google.
- Msn_Delay: guarda la demora temporal que tiene el agente del buscador MSN.
- url1: guarda la url del resultado correspondiente de Google. Tenemos también url2 y url3.
- title1: guarda el título del resultado correspondiente de Google. Tenemos también title2 y title 3.
- snippet1: guarda un fragmento del resultado correspondiente de Google. Tenemos también snippet2 y snippet3.
- title_msn1: guarda el título del resultado correspondiente de MSN. Tenemos también title_msn2 y title_msn3.
- description1: guarda la descripción del resultado correspondiente de MSN. Tenemos también description2 y description3.

- url_msn1: guarda la url del resultado correspondiente de MSN. Tenemos también url_msn2 y url_msn3.

De forma que después de resolver el problema de la adaptación de resultados, y el problema de reflejar las demoras de tiempo (cosa que ya resolvimos con la composición MarketPlace), finalmente la plantilla quedó implementada de la siguiente manera (MetaSearchTest-CSV.bpts):

```
<?xml version="1.0" encoding="UTF-8"?>
<tes:testSuite xmlns:msn="http://schemas.microsoft.com
       /MSNSearch/2005/09/fex"
     xmlns:met="http://examples.bpelunit.org/MetaSearch"
     xmlns:tes="http://www.bpelunit.org/schema/testSuite"
     xmlns:goog="http://examples.bpelunit.org/GoogleBridge">
  <tes:name>MetaSearchTest</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
  <tes:deployment>
    <tes:put name="MetaSearch" type="activebpel">
      <tes:wsdl>MetaSearch.wsdl</tes:wsdl>
      <tes:property name="BPRFile">MetaSearch.bpr</tes:property>
    </tes:put>
    <tes:partner name="Google" wsdl="GoogleBridge.wsdl"/>
    <tes:partner name="MSN" wsdl="msnsearch.wsdl"/>
  </tes:deployment>

  <tes:testCases>
    <tes:testCase name="MainTemplate" basedOn="" abstract="false"
        vary="true">
      <tes:setUp>
        <tes:dataSource type="csv" src="data.csv">
          <tes:property name="separator">,</tes:property>
        </tes:dataSource>
      </tes:setUp>
      <tes:clientTrack>
       <tes:sendReceive service="met:MetaSearch"
           port="MetaSearchPort" operation="process"
           xmlns:fex="http://schemas.microsoft.com/MSNSearch
           /2005/09/fex">
          <tes:send fault="false" delay=" $Client_Delay">
            <tes:template>
              <met:MetaSearchProcessRequest>
                <met:query> $Query</met:query>
                <met:language> $Lenguaje</met:language>
                <met:country> $Country</met:country>
                <met:maxResults> $Max_Results</met:maxResults>
              </met:MetaSearchProcessRequest>
```

```
          </tes:template>
        </tes:send>
        <tes:receive fault="false">
          <tes:condition>
           <tes:expression>
             count(met:MetaSearchProcessResponse/met:result)
           </tes:expression>
           <tes:value>number( $Count)</tes:value>
          </tes:condition>
        </tes:receive>
      </tes:sendReceive>
    </tes:clientTrack>

    <tes:partnerTrack name="Google">
      <tes:receiveSend service="goog:GoogleSearchService"
        port="GoogleSearchPort" operation="doGoogleSearch">
        <tes:send fault="false" delay=" $Google_Delay">
         <tes:template>
          <goog:SearchEngineResultList>
           <result>
             <url> $url1</url>
             <title> $title1</title>
             <snippet> $snippet1</snippet>
           </result>
           <result>
             <url> $url2</url>
             <title> $title2</title>
             <snippet> $snippet2</snippet>
           </result>
           <result>
             <url> $url3</url>
             <title> $title3</title>
             <snippet> $snippet3</snippet>
           </result>
          </goog:SearchEngineResultList>
         </tes:template>
       </tes:send>
       <tes:receive fault="false"/>
      </tes:receiveSend>
    </tes:partnerTrack>

    <tes:partnerTrack name="MSN">
      <tes:receiveSend service="fex:MSNSearchService"
        port="MSNSearchPort" operation="Search"
```

```
          xmlns:fex="http://schemas.microsoft.com/MSNSearch
          /2005/09/fex">
        <tes:send fault="false" delay="$MSN_Delay">
         <tes:template>
          <msn:SearchResponse>
           <msn:Response>
            <msn:Responses>
             <msn:SourceResponse>
              <msn:Source>$Source</msn:Source>
              <msn:Offset>$Offset</msn:Offset>
              <msn:Total>$Total</msn:Total>
              <msn:Results>
                <msn:Result>
                 <msn:Title>$title_msn1</msn:Title>
                 <msn:Description>$description1</msn:Description>
                 <msn:Url>$url_msn1</msn:Url>
                </msn:Result>
                <msn:Result>
                 <msn:Title>$title_msn2</msn:Title>
                 <msn:Description>$description2</msn:Description>
                 <msn:Url>$url_msn2</msn:Url>
                </msn:Result>
                <msn:Result>
                 <msn:Title>$title_msn3</msn:Title>
                 <msn:Description>$description3</msn:Description>
                 <msn:Url>$url_msn3</msn:Url>
                </msn:Result>
               </msn:Results>
              </msn:SourceResponse>
             </msn:Responses>
            </msn:Response>
           </msn:SearchResponse>
          </tes:template>
         </tes:send>
         <tes:receive fault="false"/>
      </tes:receiveSend>
    </tes:partnerTrack>
  </tes:testCase>
 </tes:testCases>
</tes:testSuite>
```

Y el conjunto de casos de prueba originales con los que comenzamos a trabajar son los siguientes, los cuales se encuentran recogidos en el fichero “data.csv”:

Columnas: Query, Languaje, Country, Max_Results, Count, Total, Source, Offset, Client_Delay, Google_Delay, MSN_Delay, url1, title1, snippet1, url2, title2, snippet2, url3, title3, snippet3, title_msn1, description1, url_msn1, title_msn2, description2, url_msn2, title_msn3, description3, url_msn3
Caso 1: Philipp Lahm, de, DE, 3, 3, 0, Web, 0, 0, 0, 0, http://url1google, Title1google, Snippet1google, http://url2google, Title2google, Snippet2google, http://url3google, Title3google, Snippet3google, “”, “”, “”, “”, “”, “”, “”, “”, “”
Caso 2: Philipp Lahm, de, DE, 3, 3, 3, Web, 0, 0, 0, 0, “”, “”, “”, “”, “”, “”, “”, “”, “”, Title1msn, Snippet1msn, http://url1msn, Title2msn, Snippet2msn, http://url2msn, Title3msn, Snippet3msn, http://url3msn
Caso 3.1: Philipp Lahm, de, DE, 6, 6, 3, Web, 0, 0, 0, 2, http://url1google, Title1google, Snippet1google, http://url2google, Title2google, Snippet2google, http://url3google, Title3google, Snippet3google, Title1msn, Snippet1msn, http://url1msn, Title2msn, Snippet2msn, http://url2msn, Title3msn, Snippet3msn, http://url3msn
Caso 3.2: Philipp Lahm, de, DE, 6, 6, 3, Web, 0, 0, 2, 0, http://url1google, Title1google, Snippet1google, http://url2google, Title2google, Snippet2google, http://url3google, Title3google, Snippet3google, Title1msn, Snippet1msn, http://url1msn, Title2msn, Snippet2msn, http://url2msn, Title3msn, Snippet3msn, http://url3msn
Caso 4: Philipp Lahm, de, DE, 3, 3, 3, Web, 0, 0, 0, 0, http://url1google, Title1google, Snippet1google, http://url2google, Title2google, Snippet2google, http://url3google, Title3google, Snippet3google, Title1msn, Snippet1msn, http://url1msn, Title2msn, Snippet2msn, http://url2msn, Title3msn, Snippet3msn, http://url3msn
Caso 5: Philipp Lahm, de, DE, 4, 4, 3, Web, 0, 0, 0, 0, http://url1google, Title1google, Snippet1google, “”, “”, “”, “”, “”, “”, Title1msn, Snippet1msn, http://url1msn, Title2msn, Snippet2msn, http://url2msn, Title3msn, Snippet3msn, http://url3msn
Caso 6: Philipp Lahm, de, DE, 1 ,1 ,3, Web, 0, 0, 0, 0, “”

Son seis casos de pruebas que he extraído de los ficheros BPTS originales que disponía, de forma que los he adaptado para poder aplicarlos con esta plantilla. Hay uno de los casos, el tercero, que se encuentra dividido en dos porque tiene dos demoras de tiempo distintas en los agentes Google y MSN. En el resto de casos he dejado la demora a 0, ya que no se indicaba nada en los originales.

Los resultados vacíos de los casos de prueba originales no he podido representarlos dentro de la plantilla, ya que se producía un fallo de ejecución. Así que para intentar simularlo, he recurrido a la propiedad que tiene la composición de eliminar los resultados repetidos. He utilizado los caracteres “” como un posible resultado genérico, con el cual indicamos que va haber resultados repetidos a lo largo de ese caso de prueba.

7 Automatización de las Relaciones Metamórficas

Una vez que ya hemos conseguido implementar los ficheros BPTS basados en plantillas CSV de las composiciones correspondientes y tenemos localizadas las diferencias entre los mutantes y el programa original, pasamos a diseñar las relaciones metamórficas que posteriormente vamos a implementar. En este capítulo vamos a tener un esquema común en las tres composiciones: diseño de relaciones metamórficas, obtención de algún caso de prueba a mano aplicando una de las relaciones metamórficas diseñadas a uno de los casos de prueba originales y por último, implementación de dichas relaciones.

Las aplicaciones de la composición LoanApproval y MarketPlace las implementé en un principio en código C++, pero, al pasarlas a código Java la estructura seguía siendo prácticamente la misma. Por lo que vamos a hablar sólo de las implementaciones finales que son las realizadas en código Java. Esto nos permitirá que una vez que esté todo finalizado, podamos calcular la calidad del software a través de la plataforma "Sonar" que explicaremos en el último apartado de este capítulo.

7.1. LoanApproval

A la hora de realizar los estudios de los mutantes con MuBPEL de esta composición, dijimos que existían 11 mutantes vivos después de ejecutar el conjunto de casos de prueba. De estos 11, siete se habían generado por añadir en algún lugar del mutante "//" en lugar de "/", cosa que íbamos a descartar en nuestro estudio. Por lo que realmente tenemos que centrarnos en buscar relaciones metamórficas que detecten los posibles fallos en los 4 mutantes restantes: m04-01-01.bpel, m07-01-01.bpel, m07-01-02.bpel y m07-01-03.bpel.

7.1.1. Relaciones metamórficas

Como ya vimos en el capítulo "Estudios con MuBPEL", los mutantes vivos de esta composición se debían a cambios lógicos de "<=" por "<", sumar o restar una unidad al límite monetario, y multiplicarlo por 10. Pues bien, vamos a diseñar relaciones metamórficas que estén basadas en estos posibles cambios.

Precondición1: $req\_amount1 * 10 > 10000$

MR1: $req\_amount2 = req\_amount1 * 10 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1 \ \wedge \ ap\_reply2 = accepted2$

Esta primera relación, está basada tanto en la multiplicación por 10, como en los cambios lógicos. Con ella queremos decir que si un caso de prueba 2 tiene una cantidad (amount) que es igual a la cantidad de un caso de prueba 1 multiplicada por 10, la respuesta de sus asesores es igual y la de sus aprobadores son contrarias, entonces, obtendremos el mismo resultado en ambas.

Precondición2: $req\_amount1 * 10 > 10000$
MR2: $req\_amount2 = req\_amount1 * 10 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = not(accepted1) \ \wedge \ ap\_reply2 = accepted2$

Esta segunda relación es muy parecida a la anterior, ya que es una posibilidad que también puede darse prácticamente con las mismas condiciones. Sólo que en esta ocasión, en lugar de obtener los mismos resultados en la variable "accepted", obtenemos resultados distintos.

Precondición3: $req\_amount1 * 10 <= 10000$
MR3: $req\_amount2 = req\_amount1 * 10 \ \wedge \ ap\_reply2 = ap\_reply1$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$

Cuando la cantidad del caso 1 multiplicada por 10 es menor o igual que 10000, entonces no superamos el límite monetario establecido por la composición. De esta forma, obtendremos un nuevo caso de prueba que tenga los mismos valores en el resto de variables, excepto en la cantidad monetaria que ya hemos indicado que es la misma, pero, multiplicada por 10.

Las tres relaciones metamórfica siguientes son iguales que las anteriores, sólo que en lugar de multiplicar por 10 la cantidad monetaria, la multiplicamos por 2. No es una de las diferencias que tienen los mutantes que queremos detectar, pero, su uso es muy común.

Precondición4: $req\_amount1 * 2 > 10000$
MR4: $req\_amount2 = req\_amount1 * 2 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1 \ \wedge \ ap\_reply2 = accepted2$

Precondición5: $req\_amount1 * 2 > 10000$
MR5: $req\_amount2 = req\_amount1 * 2 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = not(accepted1) \ \wedge \ ap\_reply2 = accepted2$

Precondición6: $req\_amount1 * 2 <= 10000$
MR6: $req\_amount2 = req\_amount1 * 2 \ \wedge \ ap\_reply2 = ap\_reply1$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$

Las tres relaciones metamórfica siguientes son iguales que las anteriores, sólo que en lugar de multiplicar por 2 la cantidad monetaria, le sumamos una unidad. Éste sí es uno de los cambios que poseen los mutantes.

Precondición7: $req\_amount1 + 1 > 10000$
MR7: $req\_amount2 = req\_amount1 + 1 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1 \ \wedge \ ap\_reply2 = accepted2$

Precondición8: $req\_amount1 + 1 > 10000$
MR8: $req\_amount2 = req\_amount1 + 1 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = not(accepted1) \ \wedge \ ap\_reply2 = accepted2$

Precondición9: $req\_amount1 + 1 <= 10000$
MR9: $req\_amount2 = req\_amount1 + 1 \ \wedge \ ap\_reply2 = ap\_reply1$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$

Las tres relaciones metamórfica siguientes son iguales que las anteriores, sólo que en lugar de sumar una unidad a la cantidad monetaria, le restamos una unidad. Éste también es uno de los cambios que poseen los mutantes.

Precondición10: $req\_amount1 - 1 > 10000$
MR10: $req\_amount2 = req\_amount1 - 1 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1 \ \wedge \ ap\_reply2 = accepted2$

Precondición11: $req\_amount1 - 1 > 10000$
MR11: $req\_amount2 = req\_amount1 - 1 \ \wedge \ ap\_reply2 = not(ap\_reply1)$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = not(accepted1) \ \wedge \ ap\_reply2 = accepted2$

Precondición12: $req\_amount1 - 1 <= 10000$
MR12: $req\_amount2 = req\_amount1 - 1 \ \wedge \ ap\_reply2 = ap\_reply1$
$\wedge \ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$

Una vez que ya tenemos todas las relaciones metamórficas que pueden darse, probamos a aplicar alguna de ellas en uno de los casos de prueba originales para ver si realmente obtenemos un caso de prueba correcto. Por ejemplo, vamos a aplicar la segunda relación metamórfica al siguiente caso de prueba:

(1500, false, low, true)

Como podemos ver se cumple la precondición, ya que si multiplicamos 1500*10, la cantidad final es mayor que 10000 que es el límite monetario. Por lo que podemos comenzar a aplicar la relación metamórfica 1, obteniendo lo siguiente:

(15000, false, low, false)

La cantidad queda multiplicada por 10, la respuesta del aprobador y la final es contraria a la del caso original, y la respuesta del asesor es igual. Si ahora comparamos el caso de prueba con la lógica de la composición, vemos que se trata de un caso correcto. Ya que al tratarse de una cantidad que supera el límite, entra en juego la decisión del aprobador. En este caso, el aprobador responde “true” indicando la aceptación del préstamo y por tanto, la respuesta final debe ser “true” también, cosa que se cumple. La respuesta del asesor no interviene en este caso, por lo que es un caso correcto.

Pero ahora, estudiemos el fragmento de código de la composición implicado:

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 10000)
</condition>
```

Si se produce un error, como por ejemplo, añadir un 0 más al límite monetario, el código quedaría de la siguiente manera:

```
<condition>
  (number(string($processInput.input/ns0:amount)) <= 100000)
</condition>
```

Al aplicar el caso de prueba original, en efecto, la cantidad "amount" es menor que el límite, el asesor indica que es de riesgo bajo y por tanto, el préstamo es aceptado. Ahora apliquemos el nuevo caso de prueba, de nuevo se encuentra por debajo del límite, el asesor indica que se trata de un préstamo de riesgo bajo por lo que, siguiendo la lógica de la composición, debería aceptarse. Y esto no es lo que ocurre, la respuesta final de este caso de prueba es que es denegado porque debería haber entrado en acción la figura del aprobador y no ha sido así. Por tanto, hemos detectado un posible error en la composición con este caso de prueba.

7.1.2. Implementación

Como hemos visto con esta demostración, al aplicar las relaciones metamórficas obtenemos casos de pruebas útiles a la hora de realizar estudios con la composición. Pero, uno de los problemas que implican es el costoso trabajo que pueden acarrear y la posibilidad de volver a un caso de prueba que ya habíamos diseñado anteriormente u obtenido ya por otra relación metamórfica.

Para evitar estos problemas, la automatización de esta tarea permite obtener un gran número de casos de prueba, sin tener preocupaciones de posibles casos repetidos o grandes pérdidas de tiempo en estudios que finalmente no llevan a ningún resultado relevante. La funcionalidad de la aplicación está divida en tres partes:

1. **Lectura** del fichero "data.csv" que contiene el conjunto de casos de prueba original.
2. **Generación** de nuevos casos de prueba a partir de la aplicación de las relaciones metamórficas correspondientes.
3. **Escritura** de los casos de prueba válidos en el nuevo fichero "data2.csv" que contendrá el conjunto de casos de prueba final.

Para conseguir esto, disponemos de dos clases en la aplicación:

- **LoanApprovalApplication**: se encarga de buscar el fichero "data.csv", abrirlo y guardar todos los casos de prueba que contiene en una lista. Una vez que tenemos todos los casos de prueba disponibles, vamos estudiando uno a uno y los dividimos para poder trabajar con cada uno de los valores que tienen las diferentes variables. Le aplicamos los cambios numéricos que indicamos en las relaciones metamórficas (multiplicar por 2,

por 10, sumar uno, restar uno) a la cantidad numérica de cada caso de prueba. Una vez que hemos modificado la cantidad, creamos un objeto "Caso" que aplicará las relaciones lógicas correspondientes para saber si se trata de un caso de prueba correcto. Cuando acabemos de hacer todo esto con todos los casos de prueba, se escribirán en el fichero "data2.csv" todos los casos de prueba nuevos que hayamos conseguido generar.

- **Caso**: se encarga de comprobar que los valores que recibe como un posible caso de prueba, cumplen la lógica de la composición. Si es así, lo inserta en una lista comprobando que no está ya insertado para así evitar repetidos. Si no cumple la lógica, realiza los cambios lógicos que sean necesarios para conseguir un caso de prueba válido y lo inserta, comprobándose también si ya está insertado o no. De esta forma conseguimos un nuevo caso de prueba, aunque los valores que le lleguen no sean los correctos y evitando así el realizar esta operación en vano.

7.2. MarketPlace

La composición de MarketPlace tuvo como resultados a la hora de ejecutar el conjunto de casos de prueba originales sólo un mutante vivo. No presentaba ninguna diferencia aritmética ni lógica, sólo cambiaba el orden final en el que enviaban los agentes vendedor y comprador sus respuestas. Este mutante es el m17-06-01.bpel y es en el que vamos a centrar todo nuestro estudio en esta composición.

7.2.1. Relaciones metamórficas

Para conseguir detectar fallos en este mutante, vamos a utilizar relaciones metamórficas parecidas a las de la composición anterior, que aunque no se trate de un cambio numérico o lógico el mutante generado, puede conseguir detectar algun fallo al trabajar con cantidades distintas; y también, relaciones metamórficas relacionadas con las demoras de tiempo.

Primero vamos a tratar las relaciones metamórficas más importantes a la hora de realizar modificaciones numéricas en el precio del comprador:

$$\textbf{Precondición1}: Buy\_Price1 - 1 >= Sel\_Price1 \wedge\ Buy\_Item1 = Sel\_Item1 \wedge\ Sel\_Value1 = Buy\_Value1$$
$$\textbf{MR1}: Buy\_Price2 = Buy\_Price1 - 1 \wedge\ Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Value2 = Buy\_Value1 \implies Sel\_Item2 = Sel\_Item1 \wedge\ Sel\_Price2 = Sel\_Price1 \wedge\ Sel\_Value2 = Sel\_Value1$$

Esta relación metamórfica refleja el cambio numérico en el comprador, comprobando que si sigue siendo mayor o igual que la cantidad ofrecida por el vendedor después de restarle una unidad, el nuevo caso de prueba debe tener los mismos valores que en el original (excepto la cantidad del comprador que varía).

$$\textbf{Precondición2}: Buy\_Price1 + 1 >= Sel\_Price1 \wedge\ Buy\_Item1 = Sel\_Item1 \wedge\ Sel\_Value1 = \text{“}DealFailed\text{”} \wedge\ Sel\_Value1 = Buy\_Value1$$

MR2: $Buy\_Price2 = Buy\_Price1 + 1 \wedge\ Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Value2 = \text{“}DealSuccesfull'' \implies\ Sel\_Item2 = Sel\_Item1 \wedge\ Sel\_Price2 = Sel\_Price1 \wedge\ Sel\_Value2 = Buy\_Value2$

En este caso, reflejamos la posibilidad de que al sumar una unidad en el comprador, supere o iguale el precio que indica el vendedor y que anteriormente las variables “value” eran iguales a “Deal Failed”. Si esto ocurriese, entonces todos las variables serían iguales a excepto los “value”, que indicarán ahora “Deal Succesfull” y la cantidad numérica del comprador será una unidad mayor.

Precondición3: $Buy\_Price1 * 2 >= Sel\_Price1 \wedge\ Buy\_Item1 = Sel\_Item1 \wedge\ Sel\_Value1 = \text{“}DealFailed'' \wedge\ Sel\_Value1 = Buy\_Value1$
MR3: $Buy\_Price2 = Buy\_Price1 * 2 \wedge\ Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Value2 = \text{“}DealSuccesfull'' \implies\ Sel\_Item2 = Sel\_Item1 \wedge\ Sel\_Price2 = Sel\_Price1 \wedge\ Sel\_Value2 = Buy\_Value2$

Ocurre lo mismo que en el caso anterior, sólo que esta vez en lugar de sumarle una unidad, multiplicamos la cantidad el comprador por 2.

Precondición4: $Buy\_Price1 * 10 >= Sel\_Price1 \wedge\ Buy\_Item1 = Sel\_Item1 \wedge\ Sel\_Value1 = \text{“}DealFailed'' \wedge\ Sel\_Value1 = Buy\_Value1$
MR4: $Buy\_Price2 = Buy\_Price1 * 10 \wedge\ Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Value2 = \text{“}DealSuccesfull'' \implies\ Sel\_Item2 = Sel\_Item1 \wedge\ Sel\_Price2 = Sel\_Price1 \wedge\ Sel\_Value2 = Buy\_Value2$

De nuevo la misma situación, pero, esta vez multiplicamos la cantidad del comprador por 10 en lugar de por 2.

Ahora vamos a tratar los casos más importantes a la hora de realizar cambios numéricos en el precio inicial del vendedor:

Precondición5: $Buy\_Price1 >= Sel\_Price1 - 1 \wedge\ Buy\_Item1 = Sel\_Item1 \wedge\ Sel\_Value1 = \text{“}DealFailed'' \wedge\ Sel\_Value1 = Buy\_Value1$
MR5: $Sel\_Price2 = Sel\_Price1 - 1 \wedge\ Sel\_Item2 = Sel\_Item1 \wedge\ Sel\_Value2 = \text{“}DealSuccesfull'' \implies\ Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Price2 = Buy\_Price1 \wedge\ Buy\_Value2 = Sel\_Value2$

Esta relación metamórfica nos indica que si tenemos un precio de venta inicial al cual le restamos una unidad, y el precio de compra pasa a ser igual o mayor que él (estando los valores “value” antes a “Deal Failed”), el nuevo caso de prueba que obtengamos tendrá todos los atributos iguales al caso original, excepto el precio de venta que tendrá una unidad menos y los valores “value” guardarán el valor “Deal Succesfull”.

Precondición6: $Buy\_Price1 >= Sel\_Price1 + 1 \wedge\ Buy\_Item1 = Sel\_Item1 \wedge\ Sel\_Value1 = Buy\_Value1$
MR6: $Sel\_Price2 = Sel\_Price1 + 1 \wedge\ Sel\_Item2 = Sel\_Item1 \wedge\ Sel\_Value2 = Sel\_Value1 \implies\ Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Price2 = Buy\_Price1 \wedge\ Buy\_Value2 = Buy\_Value1$

En esta ocasión, si tenemos un caso de prueba original que es satisfactorio y al sumarle una unidad más al precio de venta no conseguimos que supere el precio de compra, entonces el nuevo caso de prueba será exactamente igual al original, excepto en el precio de venta que tendrá una unidad más.

Precondición7:
$Buy\_Price1 >= Sel\_Price1*2 \wedge Buy\_Item1 = Sel\_Item1 \wedge Sel\_Value1 = Buy\_Value1$
MR7:
$Sel\_Price2 = Sel\_Price1*2 \wedge Sel\_Item2 = Sel\_Item1 \wedge Sel\_Value2 = Sel\_Value1 \implies Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Price2 = Buy\_Price1 \wedge\ Buy\_Value2 = Sel\_Value2$

Ocurre algo parecido a lo visto en el caso anterior, sólo que esta vez el precio de venta en lugar de sumarle una unidad más, la multiplicamos por 2.

Precondición8: $Buy\_Price1 >= Sel\_Price1 * 10 \wedge\ Buy\_Item1 = Sel\_Item1 \wedge\ Sel\_Value1 = Buy\_Value1$
MR8: $Sel\_Price2 = Sel\_Price1 * 10 \wedge\ Sel\_Item2 = Sel\_Item1 \wedge\ Sel\_Value2 = Sel\_Value1 \implies Buy\_Item2 = Buy\_Item1 \wedge\ Buy\_Price2 = Buy\_Price1 \wedge\ Buy\_Value2 = Sel\_Value2$

Igual que en la relación metamórfica anterior, pero, esta vez multiplicamos por 10 en lugar de por 2.

Y por último, las relaciones metamórficas que relacionan las demoras temporales de los agentes comprador y vendedor. Todos los valores que no aparezcan en las relaciones metamórficas (Price, Value, Item de cada agente) serán iguales tanto en el caso original como en el caso generado. Para simplificar su nombramiento, llamaremos a todos los valores "Resultados":

Precondición9: $Buy\_Delay1 < 10 \wedge\ Sel\_Delay1 < 10$
MR9: $Buy\_Delay2 = 10 \wedge\ Sel\_Delay2 = 10 \implies Resultados2 = Resultados1$

Con esta relación reflejamos la igualdad que debe existir entre los resultados con una demora menor al límite máximo, y los que lo igualan. Ya que mientras que no se supere ese límite máximo de tiempo establecido que es 10, los resultados deben ser iguales.

Precondición10: $Buy\_Delay1 > 0 \wedge\ Sel\_Delay1 > 0$
MR10: $Buy\_Delay2 = 0 \wedge\ Sel\_Delay2 = 0 \implies Resultados2 = Resultados1$

Con esta relación reflejamos la igualdad que debe existir entre los resultados con una demora mayor al límite mínimo, y los que lo igualan. Ya que mientras que no se tome un valor menor al límite mínimo de tiempo que es 0, los resultados deben ser iguales.

Precondición11:
$Buy\_Delay1 >= 0 \wedge\ Sel\_Delay1 >= 0 \wedge\ Buy\_Delay1 <= 10 \wedge\ Sel\_Delay1 <= 10$
MR11: $Buy\_Delay2 = Sel\_Delay1 \wedge\ Sel\_Delay2 = Buy\_Delay1 \implies Resultados2 = Resultados1$

Aquí indicamos que si las demoras de tiempo del comprador y el vendedor se encuentran dentro de los límites establecidos, a la hora de generar un nuevo caso de prueba con las demoras de tiempo intercambiadas, obtendremos los mismos resultados que en el caso original.

Precondición12:
$Buy\_Delay1 > 0 \wedge\ Sel\_Delay1 > 0 \wedge\ Buy\_Delay1 <= 10 \wedge\ Sel\_Delay1 <= 10$
MR12: $Buy\_Delay2 = Buy\_Delay1 - 1 \wedge\ Sel\_Delay2 = Sel\_Delay1 - 1 \implies$
$Resultados2 = Resultados1$

Si las demoras temporales no superan los límties máximos de tiempo y son mayores que los límites mínimos, a la hora de restarle una unidad de tiempo obtendremos los mismos resultados que en el caso de prueba original.

Precondición13:
$Buy\_Delay1 >= 0 \wedge\ Sel\_Delay1 >= 0 \wedge\ Buy\_Delay1 < 10 \wedge\ Sel\_Delay1 < 10$
MR13: $Buy\_Delay2 = Buy\_Delay1 + 1 \wedge\ Sel\_Delay2 = Sel\_Delay1 + 1 \implies$
$Resultados2 = Resultados1$

Si las demoras temporales no son menores que los límties mínimos de tiempo y no superan los límites máximos, a la hora de sumarle una unidad de tiempo obtendremos los mismos resultados que en el caso de prueba original.

Una vez que ya tenemos todas las relaciones metamórficas que pueden darse, probamos a aplicar alguna de ellas en uno de los casos de prueba originales para ver si realmente obtenemos un caso de prueba correcto. Por ejemplo, vamos a aplicar la tercera relación metamórfica y la 13ª al siguiente caso de prueba:

(Takuan, 200, Deal Failed, 0, Takuan, 120, Deal Failed, 2)

Como podemos ver se cumple la precondición3, ya que si multiplicamos 120*2, la cantidad final del comprador es mayor que el precio inicial de venta que es 200; los item del comprador y vendedor son iguales; y ambos value son iguales a “Deal Failed”. Por otro lado, también se cumple la precondición13 ya que ambos delay se encuentran dentro de los límites establecidos. Por lo que podemos comenzar a aplicar las relaciones metamórficas 1 y 13, obteniendo lo siguiente:

(Takuan, 200, Deal Succesfull, 1, Takuan, 240, Deal Succesfull, 3)

El precio que ofrece el comprador queda multiplicado por dos, los valores “value” quedan ahora a “Deal Succesfull” y las demoras de tiempo pasan a tener una unidad más de tiempo, quedando el resto de valores iguales a los que posee el caso original. Ahora si comparamos el nuevo caso de prueba con la lógica de la composición, vemos que se trata de un caso de prueba correcto, porque al ser el precio del comprador mayor o igual al del vendedor, la transacción se produce y finalmente cada uno recibe una respuesta satisfactoria, “Deal Succesfull”.

7.2.2. Implementación

Al igual que en la implementación anterior, los motivos principales por los que buscamos la automatización de la generación de los casos de prueba son el costoso trabajo que pueden acarrear y la posibilidad de volver a un caso de prueba que ya habíamos diseñado anteriormente u obtenido ya por otra relación metamórfica.

Su funcionalidad también se encuentra dividida en tres partes:

1. **Lectura** del fichero "data.csv" que contiene el conjunto de casos de prueba original.
2. **Generación** de nuevos casos de prueba a partir de la aplicación de las relaciones metamórficas correspondientes.
3. **Escritura** de los casos de prueba válidos en el nuevo fichero "data2.csv" que contendrá el conjunto de casos de prueba final.

Para conseguir esto, disponemos de dos clases en la aplicación:

- **MarketPlaceApplication**: se encarga de buscar el fichero "data.csv", abrirlo y guardar todos los casos de prueba que contiene en una lista. Una vez que tenemos todos los casos de prueba disponibles, vamos estudiando uno a uno y los dividimos para poder trabajar con cada uno de los valores que tienen las diferentes variables. Le aplicamos los cambios numéricos que indicamos en las relaciones metamórficas (multiplicar por 2, por 10, sumar uno, restar uno) al precio del vendedor y del comprador de cada caso de prueba; y los cambios correspondientes en las demoras de tiempo (límite 0 y 10, suma o resta de una unidad). Una vez que hemos modificado esto, creamos objetos "Caso" con las combinaciones de precios y tiempos que hemos indicado anteriormente en las relaciones metamórficas, y una vez dentro del objeto "Caso", se realizarán las comprobaciones lógicas correspondientes. Cuando acabemos de hacer todo esto con todos los casos de prueba, se escribirán en el fichero "data2.csv" todos los casos de prueba nuevos que hayamos conseguido generar.
- **Caso**: se encarga de comprobar que los valores que recibe como un posible caso de prueba, cumplen la lógica de la composición. Si es así, lo inserta en una lista comprobando que no está ya insertado para así evitar repetidos. Si no cumple la lógica, realiza los cambios lógicos que sean necesarios para conseguir un caso de prueba válido y lo inserta, comprobándose también si ya está insertado o no. De esta forma conseguimos un nuevo caso de prueba, aunque los valores que le lleguen no sean los correctos y evitando así el realizar esta operación en vano.

7.3. MetaSearch

Esta composición fue la que más mutantes vivos reunió después de la ejecución del conjunto de casos de prueba original. Fueron un total de 140 mutantes, pero, 40 de ellos se habían generado por la inclusión de "//" por "/" en algún lugar del mutante, cambio que no nos íbamos a centrar en estudiar. Por lo que finalmente, nuestro estudio está centrado en la detección de

fallos en 100 mutantes.

7.3.1. Relaciones metamórficas

Para conseguir detectar fallos en este conjunto tan amplio, vamos a basarnos en las diferencias que encontramos de todos ellos respecto al programa original, datos detallados en el capítulo "Estudios con MuBPEL". Estas diferencias son aritméticas, lógicas, estructurales... Lo que va a hacer que tengamos un amplio abanico de tipos de relaciones metamórficas. En cada relación vamos a incluir sólo las variables necesarias para no dificultar su comprensión, por lo que si no incluímos algún resultado o valor en particular, quiere decir que en el nuevo caso de prueba sigue teniendo el mismo valor.

Para simplificar el nombramiento de todos los resultados distintos que tenemos, tanto del agente Google como del agente MSN, vamos a llamarlos "Resultados"; y para nombrar a todos los datos que proporciona el usuario a la hora de realizar la búsqueda, vamos a llamar a todos esos datos "Consulta". Empezaremos con un conjunto, cuyas relaciones están basadas en cambios aritméticos:

Precondición1: $Consulta1 = Consulta2 \wedge\ Max\_Results1 = Count1$
MR1: $Max\_Results2 = Max\_Results1 - 1 \implies Count2 = Count1 - 1$

Con esta relación metamórfica ponemos de manifiesto parte de la lógica de la composición, además de añadir un cambio aritmético. Como la composición no permite tener un valor en Count que sea mayor a Max_Results, si tenemos un nuevo caso de prueba que realiza la misma consulta, pero, el usuario indica que quiere como salida un resultado menos, al ser Count igual que Max_Results, deberá reducir una unidad también para cumplir correctamente la lógica de la composición sin problemas.

Precondición2: $Count1 <= 3 \wedge\ Max\_Results1 >= Count1 + 3$
MR2: $Count2 = Count1 * 2 \wedge\ Max\_Results2 = Max\_Results1 \implies Resultados2 = Resultados1 * 2$

En esta ocasión, jugamos con los valores que tienen las variables "Count" y "MaxResults", ya que si Count1 es menor o igual que 3, al sumarle 3 nunca superará a 6, sólo lo puede igualar. Y éste es el valor máximo de resultados diferentes que pueden tomarse en la plantilla. Por lo que si creamos un nuevo caso de prueba que tenga el doble de Count1 y MaxResults igual, entonces tendremos el doble de resultados distintos que en el caso original.

Precondición3: $Consulta1 = Consulta2 \wedge\ Count1 = Count2$
MR3: $Max\_Results2 = Max\_Results1 * 2 \implies Resultados2 = Resultados1$

Si tenemos que en el nuevo caso de prueba las consultas son iguales, el número de resultados que nos indica las variables "Count" son iguales y el valor de Max_Results es el doble que en el caso original, entonces tendremos también los mismos Resultados, ya que nada de lo anterior debe afectarlos.

Precondición4: $Consulta1 = Consulta2 \wedge\ Count1 = Count2$
MR4: $Max\_Results2 = Max\_Results1 * 10 \implies Resultados2 = Resultados1$

Ocurre lo mismo que en el caso anterior, sólo que en esta ocasión el valor Max_Results se multiplica por 10 en lugar de por 2.

Precondición5: $Max\_Results1 > Count1 \wedge\ Count1 = Count2$
MR5: $Max\_Results2 = Max\_Results1 - 1 \implies Resultados2 = Resultados1$

Esta relación metamórfica es parecida a las anteriores, pero, en esta ocasión el valor de Max_Results es mayor que Count en el caso original. Si tomamos el mismo valor de Count en el nuevo caso de prueba y le restamos una unidad a Max_Results, entonces debemos obtener los mismos resultados, ya que Max_Results sigue siendo mayor o igual que Count, como indica la lógica de la composición.

Precondición6: $Max\_Results1 >= Count1 \wedge\ Count1 = Count2$
MR6: $Max\_Results2 = Max\_Results1 + 1 \implies Resultados2 = Resultados1$

Si aumentamos el valor de Max_Results en una unidad y no tocamos el valor que guardan las variables Count, los resultados deben seguir siendo los mismos.

Ahora vamos a explicar las relaciones metamórficas que están relacionadas con los valores de tipo cádena de la composición, como “languaje” y “country”.

Precondición7:
$(Languaje1 = \text{“”} \wedge\ Country1! = \text{“”}) \vee\ (Languaje1! = \text{“”} \wedge\ Country1 = \text{“”})$
MR7: $Languaje2 = Languaje1 \wedge\ Country2 = Country1 \implies Culture2 = Culture1$

Relación metamórfica en la cual se pone de manifiesto que si uno de los valores de “languaje” o “country” es vacío, al generar un caso de prueba con los mismos valores, debe de tener el mismo valor en “Culture”. Esto es, la cadena resultante de la la unión entre las cadenas que tienen las variables “Languaje” y “Country”.

Precondición8.1: $Languaje1! = \text{“”}$
MR8.1: $Languaje2! = Languaje1 \wedge\ Languaje2 = \text{“”} \implies Culture2! = Culture1$

Precondición8.2: $Country1! = \text{“”}$
MR8.2: $Country2! = Country1 \wedge\ Country2 = \text{“”} \implies Culture2! = Culture1$

Caso especial que impone la lógica de la composición, el cual se da cuando los valores de “languaje” o “country” de un caso original son distintos a vacío. En este caso, su campo “culture” será igual a la unión de las cadenas que guarden “language” y “country”. Por lo que si tenemos un nuevo caso de prueba, cuyos “languaje” o “country” son vacíos y por tanto, distintos a los del caso original, su campo “culture” será igual a “en-US”, cosa que nunca cumplirá el caso original.

Las dos siguientes relaciones metamórficas están relacionadas con la obtención de errores lógicos. Están pensadas para poner a prueba los capturadores de error que dispone la composición.

Precondición9: $Consulta1 = Consulta2 \wedge\ Max\_Results1 >= Count1$
MR9: $Count2 = Count1 \wedge\ Count2 > Max\_Results2 \implies Resultados2 = Resultados1$

Si generamos un caso de prueba a partir de otro original, que tiene la misma consulta y el mismo valor en "count", pero, que el valor máximo de resultados es inferior a dicho "count", no podremos obtener los mismos resultados porque la lógica de la composición no lo permite.

Precondición10: $Consulta1 = Consulta2 \wedge\ Max\_Results1 >= Count1$
MR10: $Count2 = Count1 \wedge\ Max\_Results2 = Max\_Results1 * (-1) \implies Resultados2 = Resultados1$

Con esta situación conseguimos que se produzca otro error lógico, está pensado para aquellas situaciones en las que el usuario puede incluir sin darse cuenta un signo "-" que indique que el número de resultados máximos sea negativo.

Por último, vamos a explicar cuatro relaciones metamórficas basadas en las demoras de tiempo de los agentes Google, MSN y cliente. Son parecidas a las que ya explicamos en la composición MarketPlace.

Precondición11: $Client\_Delay1 < 10 \wedge\ Google\_Delay1 < 10 \wedge\ Msn\_Delay1 < 10$
MR11: $Client\_Delay2 = 10 \wedge\ Google\_Delay2 = 10 \wedge\ Msn\_Delay2 = 10 \implies Resultados2 = Resultados1$

Con esta relación reflejamos la igualdad que debe existir entre los resultados con una demora menor al límite máximo, y los que lo igualan. Ya que mientras que no se supere ese límite máximo de tiempo establecido que es 10, los resultados deben ser iguales.

Precondición12: $Client\_Delay1 > 0 \wedge\ Google\_Delay1 > 0 \wedge\ Msn\_Delay1 > 0$
MR12: $Client\_Delay2 = 0 \wedge\ Google\_Delay2 = 0 \wedge\ Msn\_Delay2 = 0 \implies Resultados2 = Resultados1$

Con esta relación reflejamos la igualdad que debe existir entre los resultados con una demora mayor al límite mínimo, y los que lo igualan. Ya que mientras que no se tome un valor menor al límite mínimo de tiempo que es 0, los resultados deben ser iguales.

Precondición13: $Client\_Delay1 < 10 \wedge\ Google\_Delay1 < 10 \wedge\ Msn\_Delay1 < 10$
MR13: $Client\_Delay2 = Client\_Delay1 + 1 \wedge\ Google\_Delay2 = Google\_Delay1 + 1 \wedge\ Msn\_Delay2 = Msn\_Delay1 + 1 \implies Resultados2 = Resultados1$

Si las demoras temporales no superan los límties máximos de tiempo, a la hora de sumarle una unidad de tiempo obtendremos los mismos resultados que en el caso de prueba original.

Precondición14: $Client\_Delay1 > 0 \wedge\ Google\_Delay1 > 0 \wedge\ Msn\_Delay1 > 0$
MR14: $Client\_Delay2 = Client\_Delay1 - 1 \wedge\ Google\_Delay2 = Google\_Delay1 - 1 \wedge\ Msn\_Delay2 = Msn\_Delay1 - 1 \implies Resultados2 = Resultados1$

Si las demoras temporales superan los límties mínimos de tiempo, a la hora de restarle una unidad de tiempo obtendremos los mismos resultados que en el caso de prueba original.

Una vez que ya tenemos todas las relaciones metamórficas que pueden darse, probamos a aplicar alguna de ellas en uno de los casos de prueba originales para ver si realmente obtenemos un caso de prueba correcto. Por ejemplo, vamos a aplicar la 8.1 al siguiente caso:

(Philipp Lahm, de, DE, 3, 3, 0, Web, 0, 0, 0, 0, http://url1google, Title1google, Snippet1google, http://url2google, Title2google, Snippet2google, http://url3google, Title3google, Snippet3google, “”, “”, “”, “”, “”, “”, “”, “”, “”, “”)

Como podemos ver, se trata de un caso de prueba que cumple las precondiciones correspondientes: Language tiene un valor asignado. De forma que podemos aplicar la condición y obtenemos el siguiente caso de prueba:

(Philipp Lahm, , DE, 3, 3, 0, Web, 0, 0, 0, 0, http://url1google, Title1google, Snippet1google, http://url2google, Title2google, Snippet2google, http://url3google, Title3google, Snippet3google, “”, “”, “”, “”, “”, “”, “”, “”, “”, “”)

Todos los campos siguen siendo iguales, excepto Language que ahora está vacío.

Ahora vamos a estudiar el fragmento un fragmento de código de la composición, en la cual pude producirse un error:

```
<bpel:if name="Switch_10">
  <bpel:condition>
   (($inputVariable.payload/client:country != '')
      and
   ($inputVariable.payload/client:language != ''))
  </bpel:condition>
  <bpel:assign name="ConcatLangAndCountry">
   <bpel:copy>
    <bpel:from>
       concat(concat($inputVariable.payload
       /client:language, '-'), $inputVariable.payload
       /client:country)
    </bpel:from>
    <bpel:to part="parameters"
       variable="MSNSearch_Search_InputVariable">
      <bpel:query>ns2:Request/ns2:CultureInfo</bpel:query>
    </bpel:to>
   </bpel:copy>
  </bpel:assign>
  <bpel:else>
   <bpel:assign name="UseDefaulten-US">
    <bpel:copy>
     <bpel:from>'en-US'</bpel:from>
```

```
     <bpel:to part="parameters"
        variable="MSNSearch_Search_InputVariable">
      <bpel:query>ns2:Request/ns2:CultureInfo</bpel:query>
     </bpel:to>
    </bpel:copy>
   </bpel:assign>
  </bpel:else>
 </bpel:if>
```

Imaginemos que se produce el error del mutante m05-01-01.bpel, en el cual tenemos el siguiente código en ese mismo fragmento:

```
<bpel:if name="Switch_10">
  <bpel:condition>
   (($inputVariable.payload/client:country != '')
     or
   ($inputVariable.payload/client:language != ''))
  </bpel:condition>
  <bpel:assign name="ConcatLangAndCountry">
   <bpel:copy>
    <bpel:from>
       concat(concat($inputVariable.payload
       /client:language, '-'), $inputVariable.payload
       /client:country)
    </bpel:from>
    <bpel:to part="parameters"
       variable="MSNSearch_Search_InputVariable">
      <bpel:query>ns2:Request/ns2:CultureInfo</bpel:query>
    </bpel:to>
   </bpel:copy>
  </bpel:assign>
  <bpel:else>
   <bpel:assign name="UseDefaulten-US">
    <bpel:copy>
     <bpel:from>'en-US'</bpel:from>
     <bpel:to part="parameters"
        variable="MSNSearch_Search_InputVariable">
      <bpel:query>ns2:Request/ns2:CultureInfo</bpel:query>
     </bpel:to>
    </bpel:copy>
   </bpel:assign>
  </bpel:else>
 </bpel:if>
```

Si aplicamos el caso original en este fragmento modificado, vemos que la condición se cumple porque sólo se pide que uno de los campos Language o Country sea distinto de vacío. Y en

el caso original, ambos tienen un valor asignado. De forma que el campo Culture pasará a tener el valor “de-DE”.

Si ahora hacemos lo mismo con el nuevo caso de prueba, vemos que también cumple la condición, ya que el campo Country no es vacío, por lo que el campo Culture pasará a tener el valor “-DE”. Esto es un error, ya que en el fragmento original no se hubiera cumplido la condición y el campo Culture hubiera pasado a tener el valor “en-US”. Por tanto, este caso de prueba nos ha servido para encontrar un posible fallo que puede producirse en la composición.

7.3.2. Implementación

Como en las otras dos composiciones, los motivos principales por los que buscamos la automatización de la generación de los casos de prueba son el costoso trabajo que pueden acarrear y la posibilidad de volver a un caso de prueba que ya habíamos diseñado anteriormente u obtenido ya por otra relación metamórfica.

Su funcionalidad se encuentra dividida en las mismas tres partes que el resto:

1. **Lectura** del fichero “data.csv” que contiene el conjunto de casos de prueba original.
2. **Generación** de nuevos casos de prueba a partir de la aplicación de las relaciones metamórficas correspondientes.
3. **Escritura** de los casos de prueba válidos en el nuevo fichero “data2.csv” que contendrá el conjunto de casos de prueba final.

Para conseguir esto, disponemos de dos clases en la aplicación:

- **MetaSearchApplication**: se encarga de buscar el fichero “data.csv”, abrirlo y guardar todos los casos de prueba que contiene en una lista. Una vez que tenemos todos los casos de prueba disponibles, vamos estudiando uno a uno y los dividimos para poder trabajar con cada uno de los valores que tienen las diferentes variables. Le aplicamos los cambios numéricos que indicamos en las relaciones metamórficas (multiplicar por 2, por 10, sumar uno, restar uno) al valor Count o Max_Results, según corresponda, de cada caso de prueba; y los cambios correspondientes en las demoras de tiempo (límite 0 y 10, suma o resta de una unidad). Una vez que hemos modificado esto, creamos objetos “Caso” con las combinaciones de valores que hemos indicado anteriormente en las relaciones metamórficas, y una vez dentro del objeto “Caso”, se realizarán las comprobaciones lógicas correspondientes. Cuando acabemos de hacer todo esto con todos los casos de prueba, se escribirán en el fichero “data2.csv” todos los casos de prueba nuevos que hayamos conseguido generar.
- **Caso**: se encarga de comprobar que los valores que recibe como un posible caso de prueba, cumplen la lógica de la composición (número de resultados correctos en función de los valores Count y Max_Results, valor de Count respecto a Max_Results...). Si es así, lo inserta en una lista comprobando que no está ya insertado para así evitar repetidos. Si no cumple la lógica, realiza los cambios lógicos que sean necesarios para conseguir

un caso de prueba válido y lo inserta, comprobándose también si ya está insertado o no. De esta forma conseguimos un nuevo caso de prueba, aunque los valores que le lleguen no sean los correctos y evitando así el realizar esta operación en vano.

7.3.3. Casos fuera de plantilla

Durante el estudio de las relaciones metamórficas, las anteriores no fueron las únicas que desarrollé. Hay otras que también nos permite obtener otros casos de prueba, pero, no a través del fichero BPTS basado en plantillas, sino a mano. No pueden ser aplicadas en el BPTS porque implican un cambio en la estructura del fichero, algo que no podemos hacer al tratarse de una plantilla. Estas relaciones adicionales son las siguientes:

Precondición1: $Consulta1 = Consulta2$
MR1: $count(Descripciones2) = 0 \wedge\ count(Descripciones1) = count(Resultados1) \implies Resultados2 = Resultados1$

Esta relación metamórfica está relacionada con las descripciones de los resultados del agente MSN. Si tenemos un caso original, cuya consulta es igual a la del nuevo caso de prueba, y tiene tantas descripciones como resultados diferentes ha tenido, entonces los resultados del caso original y los del nuevo caso de prueba deben ser exactamente iguales. Aunque en el nuevo no se incluyan las descripciones, cosa que no debe de importar, ya que no afecta a la lógica de la composición.

Precondición2: $Consulta1 = Consulta2$
MR2: $count(Resultados1) > 1 \wedge\ diferent(Resultados1) = 1 \wedge\ count(Resultados2) = 1 \implies Count2 = Count1$

Esta relación metamórfica nos permite relacionar cambios aritméticos con los diferentes resultados que podemos tener. Si tenemos un caso original en el cual tenemos más de un resultado entre Google y MSN, pero, realmente resultados diferentes sólo tenemos 1; un nuevo caso de prueba que tenga sólo un caso de prueba y por tanto, éste sea único, entonces ambos casos de prueba tendrán el mismo valor en la variable "Count", ya que sólo tenemos un total de 1 resultado válido.

Precondición3: $Consulta1 = Consulta2 \wedge\ Max\_Results2 = Max\_Results1$
MR3: $Count2 = Count1 * 0 \implies count(Resultados2) = 0$

Si tenemos un caso de prueba original el cual tiene un Max_Results igual al del nuevo caso de prueba, y ambos realizan la misma consulta, puede darse la posibilidad de que el buscador no encuentre ningún resultado y por tanto, no tenga ninguno. Implicando de esta forma que Count valga 0.

Precondición4:
$Consulta1 = Consulta2 \wedge\ Max\_Results2 = Max\_Results1 \wedge\ Count1 > 6$
MR4: $Count2 = Count1 \wedge\ Results\_Google1 = Results\_MSN2 \implies Results\_MSN2 = Results\_Google1$

Partimos de un caso de prueba original que tiene un Count mayor que 6, y empezamos a generar un nuevo caso de prueba que realiza la misma consulta, tiene el mismo Max_Results y el mismo Count. Pues bien, si los resultados de Google del caso original son iguales a los resultados de MSN del nuevo caso de prueba, los resultados de Google del nuevo caso de prueba serán iguales a los de Google del caso original, ya que ambos casos devuelven el mismo número de resultados.

Como se puede apreciar, todas las relaciones metamórficas implican un cambio estructural en la implementación de casa caso de prueba, lo que impide su uso en la aplicación. Para poder aplicarlas y estudiar sus resultados, he desarrollado una serie de casos de prueba que se encuentran dentro del fichero **"MetaSearchTest_ExtendedAzahara.bpts"** que adjunto junto a esta memoria.

7.4. Sonar

Se trata de una plataforma para medir la calidad del software en código Java, utilizando herramientas que se encargan de extraer las métricas del código de dicho software. Entre otras cosas, nos informa de la complejidad del código, su cobertura, el número de clases que posee el software, las líneas de código, lo comentado o no que se encuentra el código, la documentación de su API correspondiente... Y nos recomienda mejoras de cómo conseguir que la calidad aumente.

Para ver de forma más gráfica la calidad del software, esta plataforma nos muestra un gráfico a la derecha de la pantalla en la cual vemos representado nuestro software a través de un rectángulo. Su tamaño va en función de las líneas de código que tiene, y el color según su calidad, va cambiando de rojo (mala calidad) a verde (buena calidad).

Nuestro código al estar dividido en tres partes, nos aparece una distribución de cada aplicación. En la Figura 7.1 podemos verlo. A la derecha tenemos el rectángulo dividido en tres partes, y cada una de ellas representa a: MetaSearch es la más grande, la superior derecha es la de LoanApproval y la inferior derecha es la de MarketPlace. Como podemos ver, tienen un color verde muy intenso que nos indica que la calidad del software es muy buena. En efecto, como indica la columna "Rules compilance" de la tabla central, la calidad del software de las tres aplicaciones es muy buena: un 100 % la aplicación de LoanApporval, un 98,7 % la de MarketPlace y un 98,1 % la de MetaSearch.

Si entramos en detalles al pulsar el nombre del software completo "MetaMorph4BPEL", nos aparece lo representado a través de la Figura 7.2. Podemos ver todos los detalles del software: líneas de código, número de clases, duplicaciones, comentarios, complejidad... Si desea estudiar más a fondo la calidad del Proyecto, en el siguiente enlace puede acceder directamente al apartado de la plataforma Sonar que está dedicado a ello [1].

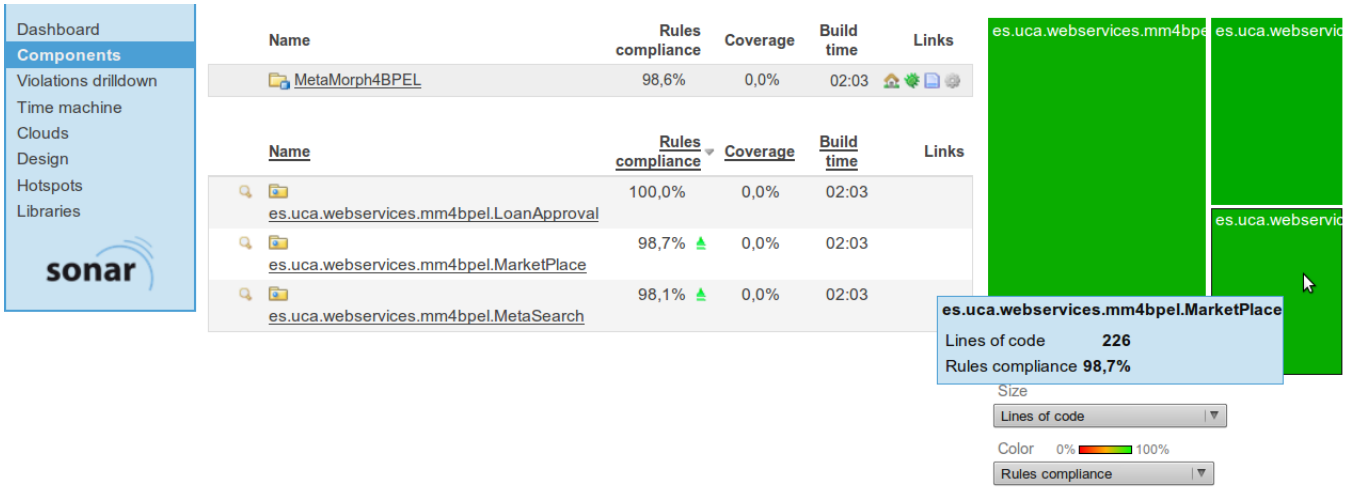

Figura 7.1: Sonar: Calidad del software general

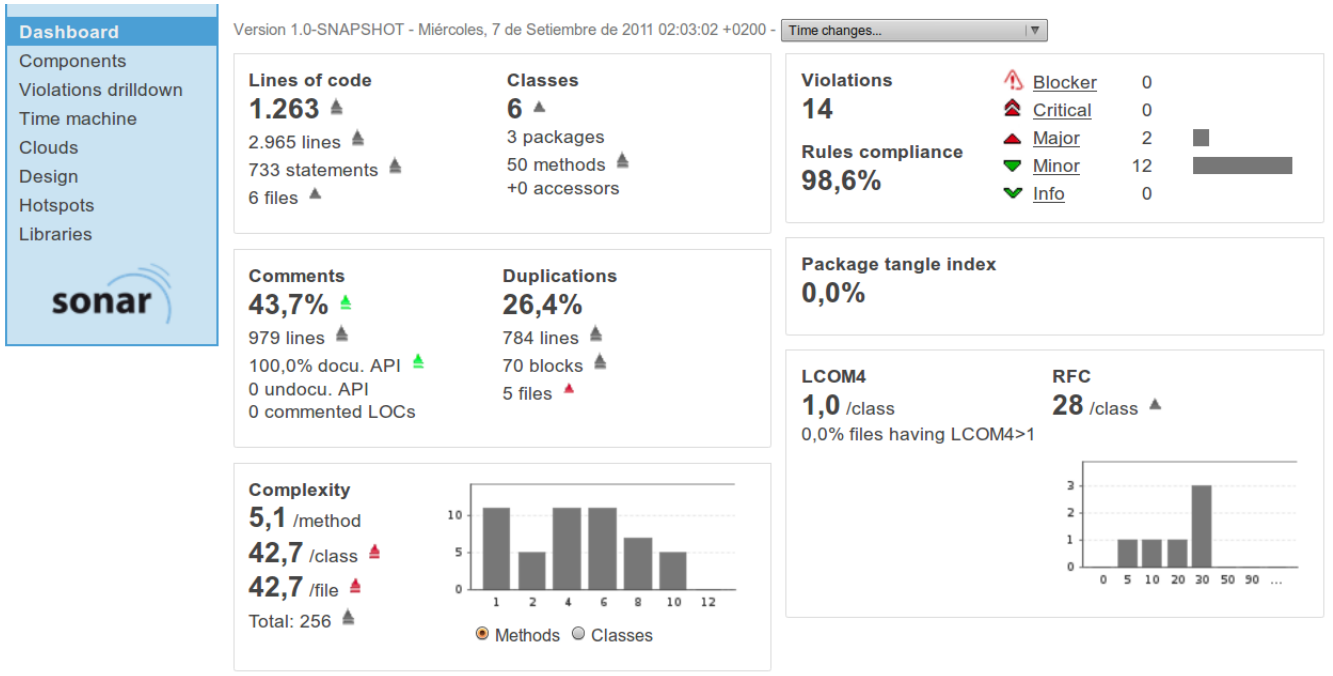

Figura 7.2: Sonar: Calidad del software detallada

8 Resultados finales

Después de obtener los nuevos casos de prueba generados gracias a las aplicaciones de cada una de las composiciones, a continuación vamos a explicar los resultados que hemos obtenido a través de la herramienta MuBPEL. A partir de ellos, indicaremos cuáles de los mutantes pueden tratarse finalmente de mutantes equivalentes o no. Sólo vamos a mostrar los resultados más importantes, de forma que aquellos que no nos aporten información relevante vamos a representarlos con puntos suspensivos(. . .).

8.1. LoanApproval

Echando la vista atrás, al comenzar el estudio de LoanApproval teniamos los siguientes números:

- Total: 93 mutantes
- Vivos: 4 mutantes (eliminando los generados por "//")
- Muertos: 82 mutantes

Por lo que nuestro objetivo era detectar errores en esos cuatro mutantes que aún seguían estando vivos. Al ejecutar la aplicación "LoanApprovalApplication" a partir del fichero "data.csv" que contenía 14 casos de prueba originales, conseguimos un nuevo fichero "data2.csv" con un total de 188 nuevos casos de prueba.

Para comprobar la calidad de estos casos de prueba, es decir, si eran buenos para detectar errores en esos cuatro mutantes, ejecutamos el programa original BPEL frente a estos nuevos casos de prueba. Y luego, comparamos los resultados obtenidos entre el programa original y los mutantes, obteniendo la siguiente salida:

m04-01-01.bpel ...0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0...
m07-01-01.bpel ...0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0...
m07-01-02.bpel ...0 0 1 1 1 0 0...0 1 1 1 0 0 0 0...
m07-01-03.bpel ...0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1...

Esto significa, como ya comentamos en el capítulo de "Fundamentos" en el apartado de MuBPEL, que al encontrarnos un "1" ese mutante está muerto al aplicarle ese caso de prueba. Y como podemos ver, en los cuatro casos hemos conseguido obtener al menos un "1", por tanto, hemos conseguido nuevos resultados relevantes.

8.2. MarketPlace

En el caso de MarketPlace al comenzar con el estudio, teniamos los siguiente mutantes:

- Total: 34 mutantes
- Vivos: 1 mutantes
- Muertos: 33 mutantes

Sólamente teniamos que centrarnos en matar a un mutante, el cual la única diferencia respecto al programa original es que mandaba antes la respuesta del agente "buyer" en lugar del agente "seller". Al igual que con la composición anterior, ejecutamos la aplicación correspondiente a ésta que es "MarketPlaceApplication", con la cual conseguimos a partir de 9 casos de prueba originales, un nuevo fichero "data2.csv" que contiene 606 nuevos casos de prueba.

Para comprobar la calidad de este conjunto de casos de prueba, ejecutamos al igual que antes el programa original BPEL frente a estos nuevos casos de prueba. Comparamos los resultados obtenidos entre el programa original y el mutante que vamos a estudiar, y obtenemos la siguiente salida:

m17-06-01.bpel `...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...`

Como podemos ver, no obtenemos ningún "1" que indique que se ha detectado algún error y por tanto, muera el mutante. Como ya hemos dicho anteriormente, la diferencia entre el programa original y el mutante es la siguiente:

```
<reply name="BuyerReply" operation="submit"
   partnerLink="buyer" portType="tns:buyerPT"
   variable="negotiationOutcome"/>

<reply name="SellerReply" operation="submit"
   partnerLink="seller" portType="tns:sellerPT"
   variable="negotiationOutcome"/>
```

Se manda antes la respuesta del comprador en lugar de la del vendedor, tal y como se indica en el programa original. Realmente, ésta no es una diferencia que afecte a la lógica de la composición, ya que todas las actividades que implican una respuesta u otra, ya se han realizado. Y en este punto lo único que queda es enviarlas. Así que, viendo que se trata de una diferencia que no afecta a la lógica de la composición, hemos pensado que se puede tratar de un **mutante equivalente**.

8.3. MetaSearch

Por último, vamos a estudiar los resultados que hemos obtenido en esta composición. Los mutantes que teníamos inicialmente son los siguientes:

- Total: 706 mutantes
- Vivos: 100 mutantes (menos los 40 generados por "//")
- Muertos: 566 mutantes

De forma, que en esta ocasión debíamos centrar nuestro estudio en los 100 mutantes que aún quedaban vivos después de aplicarles el conjunto de casos de prueba original. En este punto, nuestro estudio se divide en dos:

1. Estudio de los casos de prueba que obtenemos de la aplicación MetaSearchApplication.
2. Estudio de los casos de prueba recogidos en el fichero "MetaSearchTest_ExtendedAzahara.bpts".

Vamos a estudiar uno a uno los resultados que hemos obtenido en cada apartado, y finalmente, haremos una recopilación de los mutantes que puedan seguir vivos.

8.3.1. Casos de prueba: MetaSearchApplication

Primero, vamos a estudiar los casos de prueba que obtenemos a partir de la aplicación MetaSearchApplication. La ejecutamos y a partir de un fichero "data.csv" que contiene un conjunto de siete casos de prueba, obtenemos un nuevo fichero "data2.csv" con 1785 casos de prueba nuevos.

Para comprobar la calidad de este conjunto de casos de prueba, ejecutamos al igual que antes el programa original BPEL frente a estos nuevos casos de prueba. Comparamos los resultados obtenidos entre el programa original y los mutantes que vamos a estudiar, y obtenemos las siguientes salidas:

m01-52-01.bpel ...0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0...
m01-60-01.bpel ...0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m01-65-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-66-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0...
m01-67-01.bpel ...0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0...
m01-69-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-70-01.bpel ...0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-71-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-77-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-78-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0...
m01-79-01.bpel ...0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0...
m01-81-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-82-01.bpel ...0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0...
m01-83-01.bpel ...0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0...
m01-91-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-92-01.bpel ...0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m01-95-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...

m01-96-01.bpel ...0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m01-99-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m01-100-01.bpel ...0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m02-02-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m02-05-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0...
m02-06-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m02-06-02.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0...
m02-06-03.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0...
m02-06-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m03-01-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m03-02-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-03-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-04-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-05-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-06-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-07-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-10-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-10-05.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-12-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-13-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m04-13-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m05-01-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-06-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-06-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-06-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-06-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-09-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-09-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-10-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-10-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-11-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-11-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-11-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-12-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-12-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-13-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-13-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-14-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-14-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-19-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-19-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-20-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-20-03.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m07-20-04.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...

m11-02-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m12-01-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m12-03-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m12-07-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m12-08-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m14-01-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m14-07-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m14-10-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m17-01-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m17-06-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m17-12-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m17-23-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m20-01-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m26-01-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m26-02-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m26-03-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m26-19-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m31-01-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m31-02-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m31-03-01.bpel ...0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0...
m31-19-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m32-01-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m32-08-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m32-09-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m32-10-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m32-13-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m33-01-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m33-02-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m33-10-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m33-11-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m33-12-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m33-15-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m34-01-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m34-02-01.bpel ...0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m34-03-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m34-12-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m34-13-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m34-14-01.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...
m34-17-02.bpel ...0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0...

He marcado en negrita los mutantes que tienen al menos un "1" en la matriz de ejecución. Son un total de 32 mutantes muertos de los 100 que estabamos estudiando. A continuación, vamos a estudiar los casos de prueba que se encuentran en el fichero "MetaSearch-Test_ExtendedAzahara.bpts", pero, aplicándolos sólo a los 68 mutantes que aún siguen vivos. De esta forma tardaremos menos en realizar el estudio.

8.3.2. Casos de prueba: MetaSearchTest_ExtendedAzahara.bpts

Dentro de este fichero se encuentran implementados un total de 12 casos de prueba, los cuales no se han podido aplicar a la plantilla debido a la necesidad de modificar su estructura. Han surgido después de muchas pruebas y estudiar muy profundamente la composición MetaSearch, lo que nos ha permitido desarrollar relaciones metamórficas que han dado lugar a estos casos de prueba.

Al igual que en las pruebas anteriores, vamos a comprobar su calidad. Para ello, ejecutamos el programa original BPEL frente al conjunto de casos de prueba recogidos en este fichero BPTS. Comparamos los resultados obtenidos entre el programa original y los mutantes que vamos a estudiar, y obtenemos las siguientes salidas:

m01-65-01.bpel 0 1 0 0 0 0 0 0 0 0 0 0
m01-69-01.bpel 0 0 1 0 0 0 0 0 0 0 0 0
m01-77-01.bpel 0 1 0 0 0 0 0 0 0 0 0 0
m01-81-01.bpel 0 1 0 0 0 0 0 0 0 0 0 0
m01-91-01.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m01-95-01.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m01-99-01.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m02-02-01.bpel 0 0 1 0 0 0 0 0 0 0 0 0
m02-06-01.bpel 0 1 0 0 0 0 0 0 0 0 0 0
m02-06-04.bpel 0 1 0 0 0 0 0 0 0 0 0 0
m03-01-01.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m03-02-01.bpel 1 0 0 0 0 0 0 0 0 0 0 0
m04-03-04.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-04-04.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-05-03.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-06-02.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-07-03.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-10-03.bpel 0 0 0 1 0 0 0 0 0 0 0 0
m04-10-05.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-12-04.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-13-01.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m04-13-03.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m07-06-01.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m07-06-02.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m07-06-03.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m07-06-04.bpel 0 0 0 0 0 0 0 0 0 0 0 0
m07-09-01.bpel 0 0 0 0 0 0 0 0 0 0 1 0
m07-09-03.bpel 0 0 0 0 0 0 0 0 0 0 1 0
m07-10-01.bpel 0 0 0 0 0 0 0 0 0 1 0 0
m07-10-03.bpel 0 0 0 0 0 0 0 0 0 1 0 0

m07-11-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-11-02.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-11-04.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-12-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-12-03.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-13-02.bpel	1	0	0	0	0	0	0	0	0	0	0	0
m07-13-04.bpel	1	0	0	0	0	0	0	0	0	0	0	0
m07-14-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-14-03.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-19-02.bpel	0	0	0	1	0	0	0	0	0	0	0	0
m07-19-04.bpel	0	0	0	1	0	0	0	0	0	0	0	0
m07-20-02.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m07-20-03.bpel	0	0	0	0	0	0	0	0	0	0	0	1
m07-20-04.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m11-02-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m12-01-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m12-03-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m12-07-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m12-08-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m14-07-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m14-10-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m17-06-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m17-12-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m17-23-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m34-12-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m33-10-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m32-08-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m34-13-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m33-11-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m32-09-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m34-14-01.bpel	0	0	0	1	0	0	0	0	0	0	0	0
m33-12-01.bpel	0	0	0	1	0	0	0	0	0	0	0	0
m32-10-01.bpel	0	0	0	1	0	0	0	0	0	0	0	0
m34-17-02.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m33-15-02.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m32-13-02.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m34-03-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0
m33-02-01.bpel	0	0	0	0	0	0	0	0	0	0	0	0

En esta ocasión, con este conjunto de casos de prueba hemos conseguido matar a otros 21 mutantes. Lo que implica que entre todos los casos de prueba hemos conseguido matar a 53, reduciendo el número de mutantes vivos a 47.

8.3.3. Resultados

Como ya hemos visto, hemos conseguido detectar errores en 53 nuevos mutantes, reduciendo el número de mutantes vivos a 47. Hay algunos de estos mutantes que me gustaría explicar, ya que como comenté en el capítulo anterior en el apartado de MetaSearch, muchos de los mutantes que casualmente apunté como posibles equivalentes, siguen vivos.

Vamos a comenzar por los mutantes **m04-03-04.bpel** y **m04-04-04.bpel**, los cuales tienen la misma diferencia con el programa original:

```
<bpel:condition>
 ((boolean($done_google) = true())
 or
 (boolean($done_msn) >= true()))
</bpel:condition>
```

Compara si un valor booleano es mayor que otro, cuando este operador no es posible. Por tanto, realiza la operación "=" de nuevo. De forma que se trata de **dos mutantes equivalentes**.

El siguiente mutante que sigue vivo y que señalamos también como posible equivalente, es el **m04-07-03.bpel**. La diferencia que había respecto al programa original era la siguiente:

```
<bpel:condition>
 (($div2counter mod 2) <= 0)
</bpel:condition>
```

Se da la posibilidad de que se cumpla la actividad del condicional en aquellas situaciones cuyo resto entre div2counter y 2 sea negativo. Pero esto nunca va a suceder, ya que los posibles restos de 2 es 0 y 1. De forma que se seguirá ejecutando las mismas veces que en el programa original, porque casos que den valores negativos no habrá. Por tanto, se trata de otro **mutante equivalente**.

Por una razón parecida, el siguiente mutante también se trata de un equivalente. Es el caso de **m04-10-05.bpel**, en el cual la diferencia con el caso original era la siguiente:

```
<bpel:condition>
 (count($tempMSNResult/ns2:Description) != 0)
</bpel:condition>
```

En el caso original se cumplía la condición en aquellos casos que el número de descripciones fuese mayor que 0. Y es este caso, se cumple la condición siempre y cuando el número de descripciones sea distinto que 0. Como el número de descripciones nunca va a ser negativo, sino que sólo puede ser 0 ó mayor que 0, se ejecutará en los mismos casos de prueba que en el caso original. Por lo que se trata de otro **mutante equivalente**.

El mutante **m04-12-04.bpel**, al igual que en el caso del mutante m04-07-03.bpel, compara de nuevo si un booleano es mayor que otro. La diferencia es ésta:

```
<bpel:condition>
 ($doAdd >= true())
</bpel:condition>
```

De nuevo, esta operación no es posible realizarla entre variables booleanas, por lo que se realizará sólo la comparación de si es igual. Se cumplirá la condición las mismas veces que en el programa original, de forma que se trata de otro **mutante equivalente**.

La diferencia del mutante **m04-13-01.bpel** respecto al programa original es la siguiente:

```
<bpel:condition>
 ($added < 0.0)
</bpel:condition>
```

Compara si el valor que guarda "added" es menor que 0, cosa que nunca va a ocurrir porque el valor que le asigna la composición es siempre 0. Por tanto, esta condición nunca se va a cumplir y entrará por el camino del "else". En el caso original si se cumple la condición, se copia en la variable "result" el valor correspondiente del cliente (que realmente es la posición "result[1]"), sólo ocurre en la primera iteración ya que "added" se va incrementando en cada iteración. Y si no se cumple, se copia en la posición result[added+1]. Pero como la condición del mutante nunca se va a cumplir, siempre se va a copiar en result[added+1] independientemente del valor de "added". De forma que vamos a obtener la misma salida que el programa original, lo que indica que se trata de un **mutante equivalente**. Por esta misma razón, los mutantes **m07-20-02.bpel** y **m07-20-04.bpel** también son **equivalentes**, ya que el primero compara added con el valor -1 y el segundo con el valor -10.

La situación del mutante **m04-13-03.bpel** es parecida a la anterior:

```
<bpel:condition>
 ($added <= 0.0)
</bpel:condition>
```

En esta ocasión no compara sólo si es menor, sino que también compara si es igual, como en el caso original. Al tener la variable "added" el valor 0 como valor mínimo posible, nunca va a haber un caso de prueba que entre en la condición de ser menor que 0. Sino que sólo podrá ser como máximo igual. Por tanto, se cumplirá en las mismas ocasiones que en el caso original, obteniéndose la misma salida. Así que se trata de otro **mutante equivalente**.

Ahora vamos a estudiar las situación de **m11-02-01.bpel**, cuya diferencia con el programa original era la siguiente:

```
<forEach  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  counterName="currentResultNumber"
  name="ScanResultsForMatch"
   parallel="yes">
     ...
</forEach>
```

Se pone la propiedad parallel a “yes”, cuando en el mutante original está a “no”. Como este foreach sólo contiene una actividad que es un condicional, el cual si se cumple se realiza una asignación y en caso contrario no se realiza nada, se realice o no en paralelo, siempre se va a ejecutar sólo una actividad. De forma que vamos a obtener la misma salida que en el programa original, tratándose éste de otro **mutante equivalente**.

El siguiente mutante es **m12-01-01.bpel** y su diferencia respecto al programa original es:

```
<bpel: flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
   name="Sequence_15">
      ...
</bpel: flow>
```

Se cambia la actividad “sequence” por “flow” (a la cual se le añade las cabeceras correspondientes). Pero, si miramos el conjunto de actividades que recoge esta actividad, se trata de dos copias de datos independientes entre ellas. De forma que, se ejecute de forma secuencial o concurrente, obtendremos finalmente la misma asignación de datos y la misma salida. Por tanto, se trata de un **mutante equivalente**.

Este mutante tiene la misma diferencia que el caso anterior, pero, en otro tramo de la composición. Es el mutante **m12-03-01.bpel**:

```
<bpel: flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
   name="Sequence_3">
      ...
</bpel: flow>
```

En este caso, el conjunto de actividades que recoge son una serie de asignaciones independientes entre sí, la invocación de una actividad y luego otra asignación en la cual se refleja que se ha completado la secuencia completa sin problemas. Si se cambia la ejecución a forma concurrente, se asignarán todos los valores y se invocará la actividad, todo en el mismo momento. Si se produjera algún error, se detendría la ejecución y sería capturado por el manejador de errores; por tanto, el resultado final será el mismo que en el programa original. Así que se trata de otro **mutante equivalente**.

El mutante **m12-07-01.bpel** tiene la siguiente diferencia respecto al programa original:

```
<bpel: flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
```

```
  name="LoadGoogleResult">
    ...
</bpel: flow>
```

Nos encontramos en el programa original, con una secuencia que recoge una serie de asignaciones de valores independientes entre sí a la hora de recoger los distintos resultados de Google. Si éstas no se afectan entre ellas al cambiar de valor, el resultado debe ser el mismo si se ejecuta de forma concurrente. Por lo que se trata de otro **mutante equivalente**.

El mutante **m12-08-01.bpel** tiene la siguiente diferencia respecto al programa original:

```
<bpel: flow xmlns:f="java:es.uca.webservices.bpel.util.FuncionesXPath"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uca="http://www.uca.es/xpath/2007/11"
xmlns:conv="java:es.uca.webservices.xpath.ConversorXMLXPath"
  name="LoadMSNResult">
    ...
</bpel: flow>
```

Es la misma situación anterior, pero, en esta ocasión trabajamos con los resultados de MSN. Por la misma razón, se trata de un **mutante equivalente**.

Los mutantes **m14-07-01.bpel** y **m14-10-01.bpel** tienen la misma diferencia respecto al programa original, pero en partes distintas:

```
<bpel:if name="...">
     ...
</bpel:if>
```

Consiste en que un condicional pasa de tener la actividad “else” a no tenerla. Pero, este “else” no tenía ninguna actividad en su interior, estaba vacío. Razón por la que la salida de los mutantes y la del programa original es la misma. Asi que se trata de **dos mutantes equivalentes**.

El mutante **m17-06-01.bpel** tiene la siguiente diferencia respecto al programa original:

```
<bpel:sequence name="Sequence_3">
    ...
    <bpel:invoke inputVariable="Google_doGoogleSearch_InputVariable"
  name="Google" operation="doGoogleSearch"
  outputVariable="Google_doGoogleSearch_OutputVariable"
  partnerLink="Google" portType="ns1:GoogleSearchPort"/>
</bpel:sequence>
```

La invocación se realizaba en una zona intermedia de la secuencia, y ahora, pasa a ejecutarse como la última actividad de ella. Está actividad llama a un servicio web al cual se le envía una variable de mensaje que ya se ha completado anteriormente, tanto en el caso original como en el mutante. Por tanto, al realizarse todas las actividades necesarias antes de este “invoke”, las

salidas serán iguales. Así que se trata de otro **mutante equivalente**.

El mutante **m17-12-01.bpel** es la misma situación anterior, pero, esta vez trabaja con el agente MSN:

```
<bpel:sequence name="Sequence_4">
    ...
    <bpel:invoke inputVariable="MSNSearch_Search_InputVariable"
    name="MSNSearch" operation="Search"
    outputVariable="MSNSearch_Search_OutputVariable"
    partnerLink="MSN" portType="ns2:MSNSearchPortType"/>
</bpel:sequence>
```

Como ocurre exactamente lo mismo, pero, en esta ocasión con otras variables y la llamada del servicio web MSN, es otro **mutante equivalente**.

El siguiente mutante, **m17-23-01.bpel**, tiene la siguiente diferencia con el programa original:

```
...
<bpel:if name="NotEnoughResultsToReachMax">
...
<bpel:if name="GoogleHasMoreResults">
...
```

Cambia el orden de los "if": en el caso original se ejecutaba antes el if "GoogleHasMoreResults" y luego el if "NotEnoughResultsToReachMax". Cada uno de ellos se encarga de: asignar un valor a la variable "div2counter_final" (caso de "GoogleHasMoreResults") y de asignar un nuevo valor a la variable "div2counter_maxResults" (caso de "NotEnoughResultsToReachMax"). Como ninguno de ellos afecta al otro, se realice antes una asignación u otra, no afectará a la salida del mutante por lo que coincidirá con la salida del programa original. Por tanto, se trata de un **mutante equivalente**.

Y por último, los mutantes **m32-13-02.bpel**, **m33-15-02.bpel** y **m34-17-02.bpel** cuya diferencia con el programa original es la misma:

```
<bpel:condition>
    false()
</bpel:condition>
```

En el programa original la condición consistía en comparar si el valor de "added" era igual a 0. Como en esta ocasión el condicional no se va a cumplir nunca porque no lo permite "false()", entrará siempre por la actividad "else". Y nos encontramos de nuevo en la misma situación que en el mutante m04-13-01.bpel, que entra siempre por la actividad "else" también. Por tanto, como vamos a obtener la misma salida que en el programa original, por la misma razón que en el mutante m04-13-01.bpel, de nuevo tenemos **tres mutantes equivalentes**.

De forma que si contamos todos los mutantes equivalentes y todos los mutantes muertos, tenemos los siguientes números:

- Mutantes vivos iniciales: 100 mutantes
- Muertos: 53 mutantes
- Equivalentes: 22 mutantes
- Mutantes vivos finales: 25 mutantes

Hemos conseguido eliminar 75 mutantes entre muertos y equivalentes, dejando sólamente un total de 25 vivos.

9 Conclusiones

Este Proyecto Fin de Carrera se comenzó con la intención de avanzar en parte de la implementación de la propuesta de arquitectura que aplica pruebas metamórficas a composiciones de servicios en WS-BPEL, realizada en el grupo de investigación UCASE. Centrándonos en el estudio de tres composiciones, en las cuales debíamos detectar todos los errores posibles a través de este tipo de pruebas. A continuación, vamos a ver unos gráficos en los cuales vamos a resumir los resultados que hemos obtenido y veremos cómo ha repercutido nuestro trabajo en los datos iniciales de cada composición.

9.1. Avances en LoanApproval

En primer lugar, vamos a tratar la composición LoanApproval. Comenzamos con los siguientes datos:

- Total: 93 mutantes
- Vivos: 4 mutantes (eliminando los generados por "//")
- Muertos: 82 mutantes

Y después de aplicar todos los conceptos que hemos ido adquiriendo, las relaciones metamórficas y los casos de prueba que hemos obtenido, los resultados han sido los siguientes:

- Total: 93 mutantes
- Vivos: 0 mutantes (eliminando los generados por "//")
- Muertos: 86 mutantes

Hemos conseguido eliminar todos los mutantes vivos que eran importantes para esta composición, completando así uno de los objetivos principales de este trabajo. En la Figura 9.1 podemos ver gráficamente el antes y el después de este estudio.

9.2. Avances en MarketPlace

En el caso de MarketPlace, los datos eran estos:

- Total: 34 mutantes
- Vivos: 1 mutantes

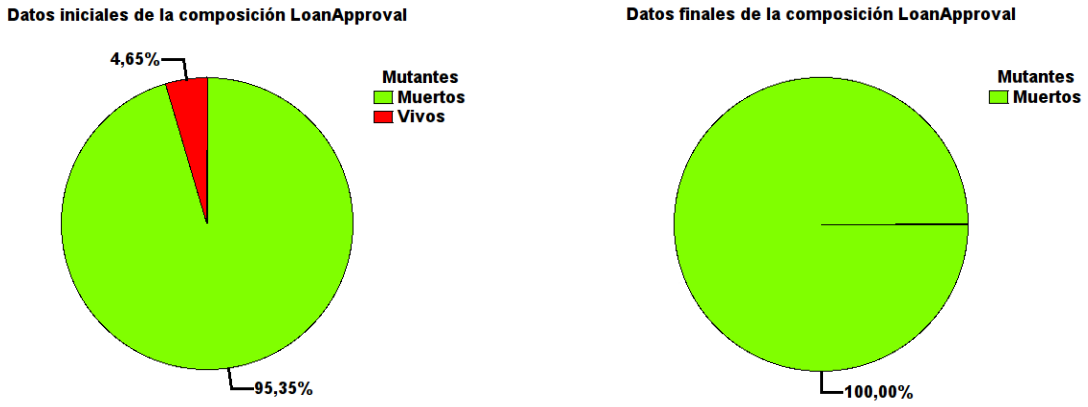

Figura 9.1: Diferencias entre datos iniciales y finales de LoanApproval

- Muertos: 33 mutantes

Y después de probar varios casos de prueba y estudiar muy detalladamente la composición, no conseguimos matar ese mutante que aún seguía vivo. Por lo que estudiamos en profundidad la diferencia que existía entre este mutante y el programa original, y vimos que no provocaba ningún cambio en la lógica de la composición. Haciendo que este mutante fuese equivalente. Por tanto, los datos finales de esta composición son:

- Total: 34 mutantes
- Vivos: 0 mutantes
- Muertos: 33 mutantes
- Equivalentes: 1 mutante

En esta composición también hemos conseguido obtener resultados relevantes, porque hemos descubierto que realmente se trataba de un mutante equivalente. Por lo que queda completo también el estudio de MarketPlace. En la Figura 9.2 podemos ver gráficamente el antes y el después de este estudio.

9.3. Avances en MetaSearch

Por último, vamos a mostrar los avances que hemos tenido en la composición MetaSearch. Los datos con los que comenzamos su estudio son los siguientes:

- Total: 706 mutantes
- Vivos: 100 mutantes (menos los 40 generados por "//")
- Muertos: 566 mutantes

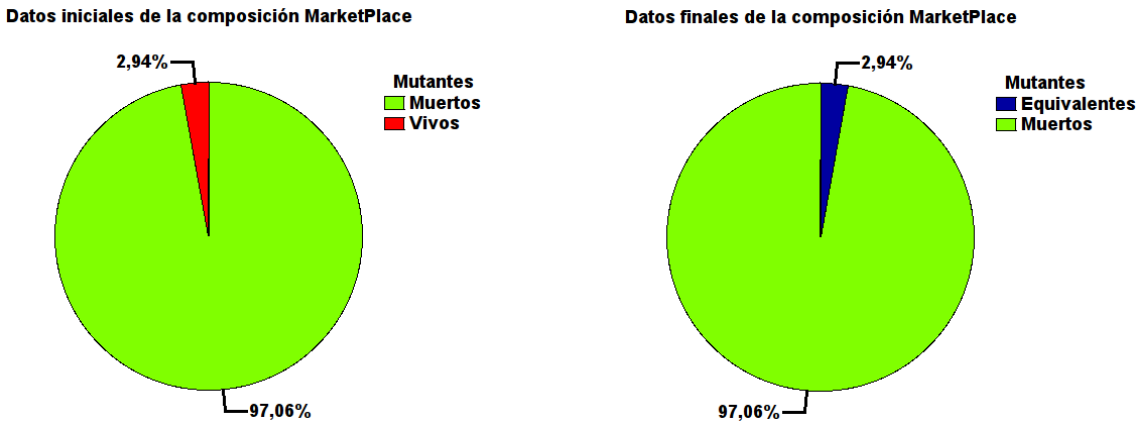

Figura 9.2: Diferencias entre datos iniciales y finales de MarketPlace

Y después de aplicar todos los casos de prueba (tanto los obtenidos por la aplicación, como los diseñados a mano por mí), de estudiar los posibles mutantes equivalentes y realizar todas las pruebas correspondientes, los nuevos resultados que hemos obtenido son los siguientes:

- Total: 706 mutantes
- Vivos: 25 mutantes
- Muertos: 619 mutantes
- Equivalentes: 22 mutantes

Hemos conseguido reducir en un 75 % el número de mutantes vivos, matando un total de 53 nuevos mutantes y clasificando 22 de ellos como equivalentes. En este caso, hemos conseguido completar parte del objetivo del estudio de esta composición, elminando gran parte de los mutantes que seguían vivos. En la Figura 9.3 podemos ver gráficamente el antes y el después de este estudio.

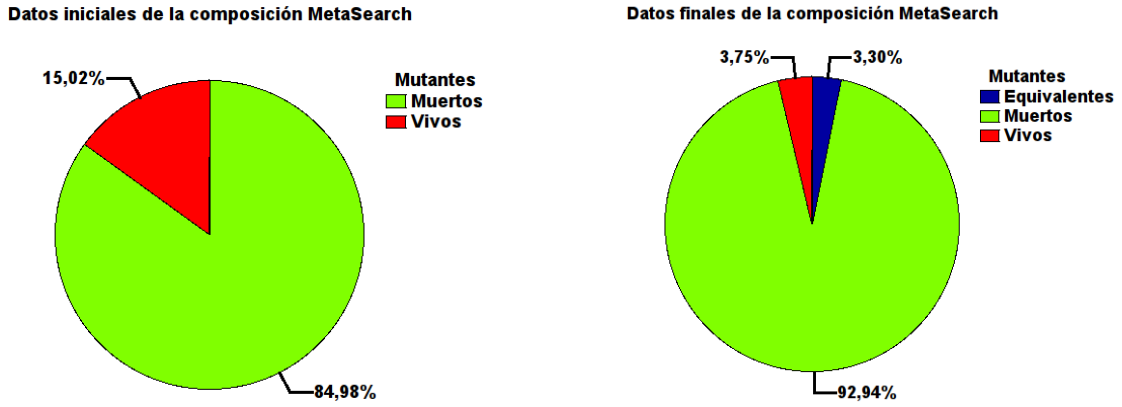

Figura 9.3: Diferencias entre datos iniciales y finales de MetaSearch

Como conclusión final, podemos decir que el conjunto de casos de prueba inicial del que disponíamos era claramente insuficiente, y esta técnica ha servido también para mejorar este conjunto de casos de pruebas. Lo que nos permite detectar más errores que antes. Podemos decir que es una buena técnica para generar de forma automática nuevos casos de prueba que detecten errores.

10 Trabajo futuro

Este trabajo de investigación tiene todavía un largo camino por delante. Algunos de los aspectos a completar consisten en continuar el trabajo que hemos empezado con este Proyecto Fin de Carrera, y otros consisten en comenzar nuevos trabajos que se encuentran muy relacionados con esto. A continuación, vamos a resaltar una serie de objetivos que se plantean como trabajo futuro.

10.1. MetaSearch: mejora del fichero BPTS basado en plantillas CSV

Uno de los objetivos principales es conseguir ampliar el tipo de casos de prueba que se pueden aplicar en la plantilla de la composición MetaSearch. De forma que no tengamos los casos de prueba repartidos por varios ficheros BPTS, sino que sólamente tengamos que añadir, eliminar o modificar los casos de prueba a través del fichero CSV correspondiente.

10.2. MetaSearch: nuevas relaciones metamórficas

Otros de los objetivos fundamentales es encontrar nuevas relaciones metamórficas que nos permitan matar a esos 25 mutantes que aún siguen vivos en esta composición y así poder encontrar posibles errores que aún no hemos encontrado. Para ello, tendremos que hacer un estudio más exhaustivo de la composición y de los distintos mutantes que tiene, para así detectar cuales son los factores que pueden matar a esos mutantes.

10.3. Nueva composición a medida de nuestros objetivos

Un nuevo trabajo que puede derivar a partir de los resultados obtenidos en éste, es el desarrollo de una composición “a medida”. Es decir, implementar una composición que realice las actividades que nosotros queramos, diseñando la lógica que más se ajuste a nuestras necesidades. Así, luego podremos aplicarle casos de prueba que estén hechos, de forma que encuentren los límites de la composición. Sería un estudio que nos permitiría trabajar de forma concreta en los aspectos que más nos interesan.

10.4. Mejorar la aplicación de las composiciones

Integrar las nuevas relaciones metamórficas que diseñemos, para que así podamos obtener nuevos casos de prueba actualizados. Con estos casos realizaríamos pruebas para comprobar si realmente son de buena calidad. Otra posible mejora en cuanto a la implementación, sería añadirle una nueva funcionalidad que le permitiera comprobar si los casos de prueba que se han diseñado e implementado nos proporcionan realmente resultados relevantes. De esta forma nos ahorraríamos tiempo realizando pruebas con MuBPEL y aumentaría la calidad del conjunto de casos de prueba final.

Manual de instalación del software

En este apéndice vamos a indicar los pasos a seguir para poder instalar en nuestro equipo todo el software necesario para realizar todas las pruebas que he ido indicando a lo largo de esta memoria. Se recomienda utilizar una distribución Ubuntu 11.04.

Los pasos a seguir para realizar la instalación son:

1. Utilizaremos el fichero "install.sh" del disco adjunto a este Proyecto Fin de Carrera que se encuentra en la carpeta "MuBPEL", o nos dirigiremos a la siguiente página web:
   https://neptuno.uca.es/redmine/projects/sources-fm/repository/changes/trunk/scripts/install.sh
   Y nos descargaremos dicho fichero.

2. Movemos el fichero de instalación al directorio que más nos convenga.

3. Ejecutamos desde ese directorio la siguiente orden:

   ```
   bash install.sh both
   ```

   Este script puede ser usado desde cualquier tipo de cuenta, pero, si necesita trabajar como administrador, necesitará privilegios "sudo".

4. Una vez que termine la instalación habremos instalado las dependecias de Gamera y Takuan, y las propias de cada uno. La que más nos importa a nosotros, MuBPEL, es una de las dependencias de GAmera. Para completarla, tendremos que salir y entrar de nuevo en nuestra sesión.

5. Un último paso antes de poder utilizar sin problemas el software instalado, es poner en funcionamiento ActiveBPEL con la siguiente orden:

   ```
   ActiveBPEL.sh start
   ```

   Aunque también puede reiniciarse utilizando:

   ```
   ActiveBPEL.sh restart
   ```

   E incluso pararlo con:

```
ActiveBPEL.sh stop
```

Una vez que termine de realizar todo esto, ya podrá utilizar todo el software que hemos indicado en este Proyecto Fin de Carrera. Si tuviese algún tipo de problema, puede dirigirse a la forja donde se encuentra la wiki de esta instalación:
https://neptuno.uca.es/redmine/projects/sources-fm/wiki
En ella encontrará todos los pasos y la información necesaria para intentar resolver dicho problema.

Manual de usuario

A continuación, vamos a explicar todas las órdenes necesarias para poder utilizar las aplicaciones de generación de casos de prueba de cada una de las composiciones, y las órdenes correspondientes a la herramienta MuBPEL para poder ejecutar dichos casos de prueba. Todos los ficheros que vamos a utilizar se encuentran dentro del disco de software que se adjunta a la memoria.

1. Obtención de casos de prueba

Dentro del disco adjunto podemos encontrar una carpeta llamada "Aplicaciones". Dentro de ella tenemos tres subcarpetas, cada una con uno de los nombres de las composiciones que hemos estudiado en este documento: "LoanApproval", "MarketPlace", "MetaSearch". En el interior de cada una podemos encontrar los siguientes ficheros:

- Fichero java que implementa la aplicación: LoanApprovalApplication.java, MarketPlaceApplication.java, MetaSearchApplication.java (uno de ellos).
- Fichero java que implementa la clase "Caso" de la composición que vamos a estudiar: Caso.java
- Fichero CSV que contiene el conjunto de casos de prueba original de la composición que vamos a estudiar: data.csv.

Para poder obtener los nuevos casos de prueba de una de ellas, copiamos al directorio que más nos convenga la carpeta de la composición que vamos a estudiar. Una vez que la hayamos copiado, nos disponemos a compilar los ficheros Java. Para ello, ejecutamos en ese directorio la siguiente orden:

```
javac *.java
```

Con ella compilaremos los ficheros ".java" que tengamos, y si surge algún problema se nos informará por terminal. Una vez que se realice esta acción, aparecerán en nuestro directorio los ficheros ".class" correspondientes a cada uno de los ficheros que hemos compilado, lo que significará que ya podemos ejecutar la aplicación sin problemas.

Si estamos trabajando la aplicación de la composición LoanApproval, tendremos que dar la siguiente orden:

```
java LoanApprovalApplication
```

En caso de estar estudiando la composición MarketPlace:

```
java MarketPlaceApplication
```

Y si nuestro estudio se centra en la composición MetaSearch:

```
java MetaSearchApplication
```

Sea cual sea la orden que demos, se nos generará un fichero "data2.csv" en el mismo directorio que nos encontramos. Y podremos utilizarlo directamente para estudiar la composición que estemos tratando. Sólo tenemos que cambiar su nombre por el de "data.csv" para que la herramienta MuBPEL pueda trabajar con él.

2. Órdenes necesarias para el uso de MuBPEL

La herramienta MuBPEL tiene una gran variedad de órdenes, pero, nosotros vamos a explicar aquí sólo las fundamentales para poder realizar las pruebas correspondientes a la composición que deseemos. En primer lugar vamos a indicar qué ficheros debemos reunir en un directorio para poder estudiar la composición correctamente:

- Fichero BPEL en el cual se encuentra implementada la composición a estudiar.
- Fichero BPTS en el cual se encuentran los diversos casos de prueba que vamos a probar.
- Fichero WSDL con las implementaciones de los servicios web.
- Fichero "build.xml" se encarga de controlar todos los ficheros a la hora de trabajar con ellos.

Dentro del disco adjunto podemos encontrar una carpeta llamada "Composiciones". Dentro de ella tenemos tres subcarpetas, cada una con uno de los nombres de las composiciones que hemos estudiado en este documento: "LoanApproval", "MarketPlace", "MetaSearch". Y cada una de ellas contiene los ficheros necesarios para poder realizar todas las órdenes que se van a explicar a continuación y que se han utilizado a lo largo de este documento.

En nuestro caso, pueden darse dos situaciones diferentes a la hora de utilizar MuBPEL:

1. Vamos a trabajar con un fichero BPTS basado en plantillas CSV.
2. Vamos a trabajar con un fichero BPTS que tiene implementado un conjunto de casos de prueba.

Existe una pequeña diferencia entre estas dos situaciones. Y consiste en que en el primer caso, además de todos los ficheros que hemos nombrado anteriormente, se debe colocar también el fichero "data.csv" en el cual vamos a recoger todos los casos de prueba que vamos a probar.

Una vez que tengamos todos los ficheros, nos disponemos a ejecutar las órdenes de MuBPEL. Se deben realizar en el siguiente orden para que no haya ningún tipo de problema:

1. Se analiza el programa WS-BPEL, obteniéndose la lista de operadores de mutación que son aplicables y las localizaciones donde se pueden aplicar.

   ```
   mubpel analyze bpel
   ```

   También puede guardarse la salida en un fichero ".txt" para tener estos resultados más a mano:

   ```
   mubpel analyze bpel > operadores.txt
   ```

2. Se generan todos los mutantes posibles.

   ```
   mubpel applyall bpel
   ```

3. Se ejecuta el programa original frente al conjunto de casos de prueba, guardando la salida en un fichero xml.

   ```
   mubpel run bpts bpel > salida.xml
   ```

4. Con la siguiente orden comparamos el resultado de ejecutar el programa original con el resultado de cada mutante para cada caso de prueba, obteniendo de esta manera una matriz de ejecución. Podemos ejecutarla de maneras diferentes, pero, las dos más importantes pueden ser las siguientes:

   ```
   mubpel compare bpts bpel salida.xml mutanteX.bpel...
   ```

   Esta forma compara los resultados entre el programa original y los mutantes hasta que encuentra una diferencia, luego deja de comparar para agilizar el estudio.

   ```
   mubpel compare -k bpts bpel salida.xml mutanteX.bpel...
   ```

   Obliga a MuBPEL a comparar todos los resultados entre los mutantes y el programa original, halla o no habido ya una diferencia en casos de prueba anteriores.

Después de realizar todas estas órdenes aparecerá por pantalla la matriz de ejecución de la composición, generada por la orden "compare" y es lo que nos servirá para estudiar si los mutantes siguen vivos o muertos.

Si quiere conocer todas las órdenes que dispone la herramienta MuBPEL o necesita ayuda de algún tipo, ejecute la orden "help" que le mostrará por pantalla las diversas órdenes que dispone:

```
mubpel -help
```

Y si quiere conocer más a fondo alguno de los comandos de MuBPEL, la siguiente orden le mostrará por pantalla toda la información que hay de él:

```
mubpel comando -help
```

Bibliografía

[1] Detalles del Proyecto Fin de Carrera dentro de la plataforma Sonar, "https://neptuno.uca.es/sonar/components/index/592".

[2] ActiveVOS. ActiveBPEL WS-BPEL and BPEL4WS engine. http://sourceforge.net/search/?q=ActiveBPEL, October 2009.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 402–411. ACM Press, 2005.

[4] M. Bozkurt, M. Harman, and Y. Hassoun. TR-10-01: testing web services: A survey. Technical Report TR-10-01, King's College, London, 2010.

[5] M. C. Castro Cabrera, M. A. Camacho Magriñan, I. Medina Bulo, and M. Palomo Duarte. Una arquitectura basada en pruebas metamórficas para composiciones de servicios WS-BPEL. In *Actas de las VII Jornadas de Ciencia e Ingeniería de Servicios*, A Coruña, Spain, September 2011.

[6] T. Y. Chen. Metamorphic testing: A new approach for generating next test cases. *HKUSTCS98-01*, 1998.

[7] T. Y. Chen, Jianqiang Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: A case study. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 327–333. IEEE Computer Society, 2002.

[8] T. Y Chen, D. H Huang, T. H Tse, and Z. Q Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583, 2004.

[9] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. GAmera: An automatic mutant generation system for WS-BPEL compositions. In *ECOWS 2009: Seventh IEEE European Conference on Web Services*, pages 97–106. IEEE Computer Society, 2009.

[10] J. J. Domínguez Jiménez, A. Estero Botaro, I. Medina Bulo, M. Palomo Duarte, and F. Palomo Lozano. El reto de los servicios web para el software libre. In *Proceedings of the FLOSS International Conference 2007*, pages 117–132. Servicio de Publicaciones de la Universidad de Cádiz, March 2007.

[11] A. García Domínguez. Test case templates in bpelunit, 2011.

[12] A. García Domínguez, I. Medina Bulo, and M. Marcos Bárcena. Inference of performance constraints in Web Service composition models. In *Proceedings of the 2nd International Workshop on Model-Driven Service Engineering*, Málaga, Spain, July 2010.

[13] A. Gotlieb and B. Botella. Automated metamorphic testing. *Computer Software and Applications Conference, Annual International*, 0:34–40, 2003.

[14] G. Di Guglielmo and the contributing authors. Mutation testing online, febrero 2010.

[15] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering*, 2010.

[16] P. Mayer and D. Lübke. Towards a BPEL unit testing framework. In *TAV-WEB'06: Proceedings of the 2006 workshop on Testing, Analysis, and Verification of Web Services and Applications*, pages 33–42. ACM, 2006.

[17] C. Murphy, G. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. In *Proc. of the 20th international conference on software engineering and knowledge engineering (SEKE)*, pages 867–872, 2008.

[18] C. Murphy, K. Shen, and G. Kaiser. Using JML runtime assertion checking to automate metamorphic testing in applications without test oracles. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 436–445, 2009.

[19] OASIS. Web Services Business Process Execution Language 2.0. Organization for the Advancement of Structured Information Standards, 2007.

[20] M. Palomo-Duarte, A. García-Domínguez, I. Medina-Bulo, A. Álvarez-Ayllón, and J. Santacruz. Takuan: a tool for WS-BPEL composition testing using dynamic invariant generation. In B. Benatallah et al., editor, *ICWE*, volume 6189 of *Lecture Notes in Computer Science*, pages 532–535. Springer, 2010.

[21] E. Weyuker. On testing Non-Testable programs. *The Computer Journal*, 25(4):465–470, November 1982.

[22] Y. Zheng, J. Zhou, and P. Krause. An automatic test case generation framework for web services. *Journal of software*, 2(3):64–77, 2007.

[23] Z. Q Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*. Software Engineers Association, 2004.